5-1-2008

Group-Aware Stream Filtering

Ming Li
Dartmouth College

GROUP-AWARE STREAM FILTERING

Dartmouth Technical Report TR2008-621

A Dissertation

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Ming Li

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 2008

Examining Committee:

(chair) ***David Kotz***

Andrew Campbell

Paul Thompson

Apratim Purakayastha

Charles K. Barlowe, Ph.D.
Dean of Graduate Studies

Abstract

Recent years have witnessed a new class of monitoring applications that need to continuously collect information from remote data sources. Those data sources, such as web click-streams, stock quotes, and sensor data, are often characterized as fast-rate high-volume "streams". Distributed stream-processing systems are thus designed to efficiently use system resources to serve the data-acquisition needs of the applications. Most of the state-of-the-art stream-processing systems assume an Ethernet-based network whose bandwidth is abundant, and focus on mechanisms to save computational power and memory. For applications involving wireless networks, particularly multi-hop mesh networks, we recognize that the most limiting factor in efficiently processing streams lies in the network's highly constrained bandwidth. Hence, this dissertation proposes a *group-aware stream filtering* approach that saves bandwidth at the cost of increased CPU time, for low-bandwidth data-streaming systems.

This approach, used together with multicasting, exploits two overlooked properties of monitoring applications: 1) many of them can tolerate some degree of "slack" in their data quality requirements, and 2) there may exist multiple subsets of the source data satisfying the quality needs of an application. We can thus choose the "best alternative" subset for each application to maximize the data overlap within the group to best benefit from multicasting. After proving the problem NP-hard, we introduce a suite of heuristics-based algorithms that ensure data quality, specifically data granularity and timeliness, in addition to preserving network bandwidth.

Our framework for group-aware stream filtering is extensible and supports a diverse range of filtering needs of monitoring applications. We evaluate this approach with a prototype system based on real-world data sets. The results show that quality-managed group-aware filtering is effective in trading CPU time for bandwidth savings, compared with self-interested stream filtering. We also evaluate the effect of each algorithm on temporal freshness of the data. Finally, we discuss other application realms that might benefit from group-aware stream filtering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent years have seen the rise of distributed systems that support applications that continuously collect, aggregate and disseminate data from information sources across a network. Those data sources, such as click-streams, stock quotes, and sensor data, are often characterized as fast-rate high-volume "streams" [5]. The systems are thus called *distributed stream-processing systems*. Applications that feed on the high-volume data sources are usually *monitoring applications* coming from many different domains; they range from RFID-based inventory management, pipeline monitoring for civil engineering, real-time stock-price analysis, mining of web click-streams, habitat monitoring, to vital-sign monitoring and medical triage. It is well recognized that the traditional store-index-and-then-process model is no longer fit for processing streaming data that have a high data rate and may be unbounded in length. Hence, stream processing has attracted much attention in recent years from the database and systems research communities.

1.1 Research challenges and the state of the art

One of the core research challenges for data stream systems is to manage the data load [61], both in data processing and in the network that transports the data. Since the purpose of many monitoring applications is to get the "big picture" of a situation, processing all data for accurate results may not be necessary for the desired accuracy. Thus, much of the research effort has been in designing suitable data structures and algorithms for stream processing that produces fast and approximate results: this includes 1) data-shedding algorithms, such as sampling algorithms, that can adaptively reduce data according to the down-stream operators' processing capacity, 2) synopsis data structures that capture the most important features of data streams with a manageable size, and 3) approximation algorithms to alleviate the "blocking" effect of operators, such as top-k and table-join, that typically need to get the entire data input before generating any output and thus block the data flow for infinite data streams. To reduce data load in the network, **in-network processing** pushes data-reducing operators, such as filters, close to the data source to reduce the data volume sent to the applications. It has a proven edge over the application-side-only processing in terms of system scalability [5].

These ideas of approximate processing for distributed data streams have resulted in several major stream-processing systems in the literature. Stanford's STREAM [2, 3], from a database perspective, extends the declarative SQL language for continuous queries, and dynamically generates query plans for approximate stream processing. For table-join operation and other blocking operators, they introduce the concept of sliding "windows" to segment a time series into finite data batches and compute approximate results on each batch. Berkeley's TelegraphCQ [19] treats data stream queries and data in the same way and dynamically routes and matches data with queries, tuple by tuple. MIT's Aurora/Borealis [7] and Dartmouth's Solar [21] allow applications to explicitly use data-fusion graphs with

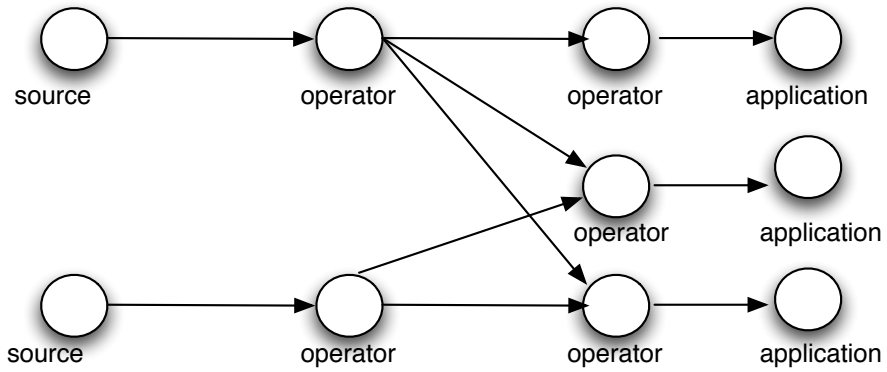


Figure 1.1: General work-flow graph in a stream-processing system

user-defined aggregates for data processing. The data-fusion graph for an application is a tree rooted at an application with data sources as the leaves, and operators as intermediate nodes; multiple applications may share data sources or operators and thus we can use a circle-and-arrow acyclic graph (such as that shown in Figure 1.1) to represent a general structure of work flows for data stream processing for multiple applications. In this thesis, we assume such a work-flow for stream processing. In the rest of the thesis, we consider any data-sharing junctures in a stream-processing work flow "data sources". Data sources may include operators as well as the root sources that start the work flow. We call the nodes where data sources reside "source nodes".

Many of the stream-processing systems consider CPU and memory the most challenged system resources for data stream processing. They design CPU-efficient and memory-efficient algorithms for complex operators such as table-join, and design load balancing and operator migration algorithms for long-running programs. For example, NiagraCQ [23] considers CPU and memory optimization for multiple continuous queries by sharing the computation of common sub-expressions among the query predicates.

For in-network processing of high-volume data streams, it is important to manage network load efficiently to make the stream-processing system scalable to the number of the

applications it serves. There are two main strategies for saving bandwidth. First, it is desirable to monitor and estimate the selectivity of operators and deploy those with high selectivity close to data sources to minimize the overall bandwidth consumption. Such in-network deployment can save overall bandwidth. Secondly, we can use multicast to transport the output of an operator to all its remote downstream operators to eliminate redundant data at network links.

In this thesis, we consider a distributed stream-processing system that assumes saving network bandwidth for data dissemination is far more critical than saving CPU power or memory. Let us consider an emergency-response scenario where data traverse multi-hop wireless mesh networks that are formed by routers on fire trucks, police cars and ambulances at the scene. The effective bandwidth in a wireless mesh network is typically much smaller than its link capacity, and smaller than its wired counterpart. High-volume source data, such as from environmental sensors or location-tracking systems, continuously flow to remote situation-awareness applications simultaneously. For data-intensive applications such as fire prediction, medical triage, and personnel tracking, there is a clear disparity between the high-resolution data-acquisition needs and limited network bandwidth. Thus, saving CPU power and memory cost of the processing on sufficiently powerful computers is less critical than preserving bandwidth for dissemination, which contrasts with the goals of many state-of-the-art stream-processing systems that assume an Internet-based network connection. In this thesis, we use a novel "group-aware" filtering approach, used in conjunction with multicast, to spend CPU time to save network bandwidth.

1.2 Our approach

Our group-aware stream filtering approach hinges on the fact that many applications that subscribe to the same data source may need different portions of the data. This may be

due to applications' different requirements on data granularity, varying capacity of their data-receiving nodes, or other factors. It thus reduces bandwidth consumption to deploy a filter on the source node that chooses only the data important to its corresponding application before transporting the data. Further, if we multiplex and then multicast the outputs of those source-sharing filters, we can eliminate redundant communication in the network. Note, the output of a filter is a subset of the source data; our filters do data selection only, as opposed to "filters" mentioned in other contexts [71, 30] that may also include data transformers or aggregators. Such a combination of data filtering and multicasting is illustrated in Figure 1.2: two applications, A and B, share the same data source D, but each application's filter selects a different subset on the source node. The multicast protocol allows us to label each tuple with the list of the applications that should receive that tuple; thus each tuple is transmitted at most once on any link. In this setting, the objective of our group-aware stream filtering is to make each filter "group-aware" such that the combined outputs of the filters is minimized in size, while still satisfying the applications' needs simultaneously. This reduced total output can better reap the benefit of multicast.

The key characteristics of a group-aware filter is its capability to select an output from multiple quality-equivalent potential outputs that satisfy the requirements of its application. For many exploratory monitoring applications, data streams provide a series of state updates about an interested environment, and filters that compress the state updates based on applications' data granularity are of special interest for saving network bandwidth. Those compressing filters can be turned group-aware based on two overlooked, yet important, properties of the applications: 1) many applications can tolerate some degree of "slack" in their data quality requirements, and 2) there may exist multiple subsets of the source data satisfying the quality needs of an application. We can thus make filters group-aware and choose the "best alternative" subset for each application, maximizing the data overlap within the group to best benefit from multicasting.

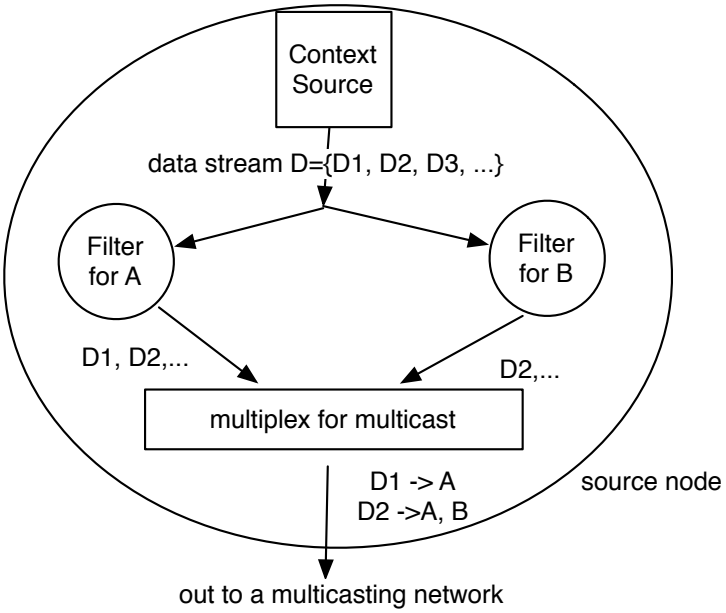


Figure 1.2: Filtering for multicasting.

Data granularity measures the level of details of domain-specific features embedded in a source data stream. Using aggressive filtering can reduce the data bandwidth consumed by applications, but it may also degrade the data granularity. When the resulting bandwidth consumption of the filter-then-multicast approach reaches the limit of the network, rather than resorting to more aggressive filters that may severely reduce the data quality, we propose a *group-aware stream filtering* approach, combined with multicast, to explore further bandwidth-saving opportunities within the same level of data granularity required by applications. Figure 1.3 shows the trade-off between bandwidth consumption and data granularity in data dissemination. When the bandwidth capacity is low, as shown, a group-aware solution can squeeze the data into the network pipe while maintaining the same level of targeted data granularity.

Note that this dissertation focuses solely on the data-reducing aspect of group-aware stream filtering; we restrain ourselves from investigating how its performance adapts to the

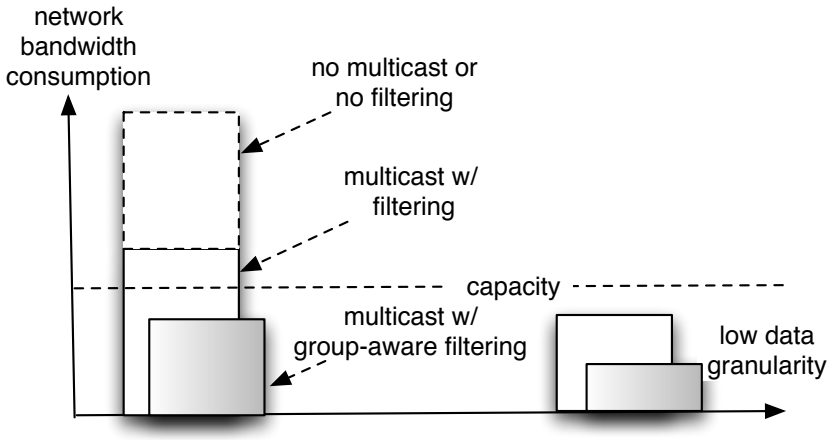


Figure 1.3: A trade-off in data filtering.

network dynamics and how the network dynamics affects the performance of group-aware filtering. Yet, high dynamics of network condition in wireless mesh networks is another well-recognized challenge faced by stream-processing systems. Furthermore, we do not investigate the specifics of the multicast protocol or leave the investigation of network aspect of group-aware filtering to future work.

1.2.1 Contributions

This dissertation makes the following contributions.

- It recognizes that, in scalable stream-processing systems with low-bandwidth wireless networks, saving bandwidth is the most important goal. Based on key observations of properties of monitoring applications, we propose a group-aware stream filtering approach to this end.

- It thoroughly treats the group-aware stream filtering problem by formally proving its NP-hardness, and by providing a general framework encompassing a suite of heuristics-based algorithms for the problem.

- Our algorithms ensure data timeliness and data granularity, in addition to the goal of preserving network bandwidth. Thus our approach is quality-managed.

- Group-aware filtering supports a diverse range of data-selection operators. Although we use delta-compression filters throughout the dissertation as examples for illustration purposes, we rigorously address the extensibility of our framework to support filters beyond delta compression, such as sampling filters. We also discuss the possibility of adopting group-aware stream filtering for uses other than saving network bandwidth.

- We built a prototype system for evaluation. Our results, based on real-world data sets, show that our group-aware filtering method can effectively save bandwidth (with low CPU overhead) when compared with self-interested filtering. We also evaluate the effect of each algorithm on time freshness of the data.

We also explain some of the limitations of group-aware filtering. For example, the complexity and overhead of using group-aware filtering may make it unfit for some application scenarios. Also, our framework is not a full-featured framework, as it does not address the security vulnerability it introduces to the system. We provide some directions for future extensions.

In Chapter 2, we overview the group-aware stream filtering approach. In Chapter 3, we introduce the mechanisms used for enforcing data timeliness in group-aware stream filtering. In Chapter 4, we evaluate our approach with a prototype system. In Chapter 5, we show that our framework is extensible to support diverse filters and can be used for scenarios other than saving network bandwidth. In Chapter 6, we conclude the thesis with discussion of the limitations of the group-aware filtering approach and some future work.

Chapter 2

Group-aware Stream Filtering

In this chapter we overview group-aware stream filtering by first introducing two key observations of the applications that motivate the work. Then we formally define the core problem of group-aware stream filtering. To deal with a potentially infinite data stream, we introduce a methodology that segments the source data stream when applying filtering and we prove that it preserves the solution's optimality for the overall stream. Proving the problem NP-hard, we develop two heuristics-based filtering algorithms that solve the problem approximately.

2.1 Two motivating observations

Our group-aware stream filtering approach is based on two key observations about the data-quality requirements of monitoring applications that stream-processing systems serve. Data quality is normally measured as the *accuracy*, *granularity*, *timeliness*, and *completeness* of the data [34]. Implications of data quality at different parts of the data-acquisition process may be different. For filtering, ensuring accuracy and completeness may mean that filters must not tamper with the input data (enforcing accuracy), and that filters must output all

tuples in the input data stream that satisfy applications' needs (enforcing completeness). We assume that the chosen filters can always ensure these two qualities. The filters' main job is to select an appropriate subset of input data that meets the applications' data granularity requirement. For example, an application would like to get a temperature reading of a place whenever the reading has changed by n degrees. This n-degree data granularity requirement can be enforced by a *Delta-Compression* (DC) filter that compresses the streaming data at "delta", in this case n degree, intervals. The higher the data granularity is, in the case of DC filters, the lower the "delta" interval, and the more output a filter should normally produce. Data granularity thus directly affects bandwidth consumption. The timeliness requirement at the filter can be measured by the amount of delay introduced by filtering. The faster a filter processes and outputs the data, the more timely is the data delivered to applications.

Delta-compression filters are data-reduction mechanisms commonly used to help exploratory applications to select most relevant state updates from high-volume status-reporting data streams over a low-bandwidth network. Delta compression has also been applied for distributing updated versions of software or synchronizing files between different accounts and devices especially over a networked environment where data and content are widely replicated and frequently modified. This dissertation uses this type of filters as illustrating examples. Note that group-aware filtering is general and applicable to many other types of filters as we show in Chapter 5.

2.1.1 First observation

Monitoring applications may tolerate some degree of "slack" in their data quality. Consider a temperature source and delta-compression filtering, for example. Given a time-

ordered nine-tuple sequence from the source, $\{0, 35, 29, 45, 50, 59, 80, 97, 100\}$,[1] the output that satisfies compression at 50-unit granularity is $\{0, 50, 100\}$. We recognize that applications may find it harmless to tolerate a small deviation from the ideal compression granularity in the output. For instance, the application may be able to tolerate a maximum of 10-degree "slack" with regard to its ideal 50-degree granularity requirement. We denote such filters as a *(slack, delta) Delta-Compression filter,* which selects data at delta-unit with slack-unit of quality deviation.

2.1.2 Second observation

There may exist more than one sequence from a data source that can satisfy an application's approximate quality requirements. In the previous example, if the application tolerates a maximum of 10-degree slack in the 50-degree compression granularity, it is easy to validate that the following sequences each satisfy the approximate granularity requirements as well: $\{0, 45, 100\}$, $\{0, 59, 100\}$, $\{0, 50, 97\}$, $\{0, 45, 97\}$, $\{0, 59, 97\}$, as 45, 59 are close-by tuples within 10-degree deviation from "ideal" output 50 after initial output 0, and 97 is close to the "ideal" output 100 as a third output.

2.1.3 Group-awareness

Let us call the above delta-compression application A. Suppose application B shares the same source as A and tolerates a maximum of 5-degree slack in a 40-degree compression granularity. By the above definitions, it is easy to validate that the following sequences satisfy B's requirements: $\{0, 45, 97\}$, $\{0, 50, 97\}$, $\{0, 50, 100\}$, $\{0, 45, 100\}$.

Individually, A may choose $\{0, 50, 100\}$ as its output; B may choose $\{0, 45, 97\}$ as its output; there are thus 5 tuples to output when multiplexing the output streams for mul-

[1] Here we represent each tuple as a single integer; in reality, each tuple may have several fields, but for simplicity we represent each by the value of its "temperature" field since it is that field that is used for filtering.

ticasting. If A and B are aware of each other's filtering needs, and both decide on, say, $\{0, 50, 97\}$ as their individual output, then only three tuples need to be multicast to A and B to satisfy both filtering requirements. In effect, the "group-awareness" reduces the bandwidth demand by two tuples.

2.2 Problem definition

Before formally defining the core problem in group-aware stream filtering, we first introduce the preliminaries, and the notion of "candidate sets" important for the problem.

2.2.1 Preliminaries

Here we introduce our assumptions and background information about the data stream systems that this dissertation concerns.

We assume a wireless network of commodity computers that form an overlay infrastructure for distributed data stream processing and dissemination. Each node, unlike a typical sensor node, is powered sufficiently and has fast CPU to support complex computations. We can imagine such an overlay network can be formed by nodes embedded in the fire trucks, police cars and ambulances that have arrived on a disaster scene to serve monitoring applications for emergency response. We assume the network bandwidth is much smaller than its link capacity due to multi-hop routing and signal interference, thus bandwidth is the most constrained system resource.

The overlay infrastructure hosts a collection of data sources that advertise their data via *source proxies*, and remote applications can subscribe to one or more of the sources via *sink proxies*. We assume that the system deploys in-network processing operators for each application and the data flows from sources to sinks through an acyclic graph. Each operator gets input from its *upstream* operator(s) or proxy (proxies), and provides output for its

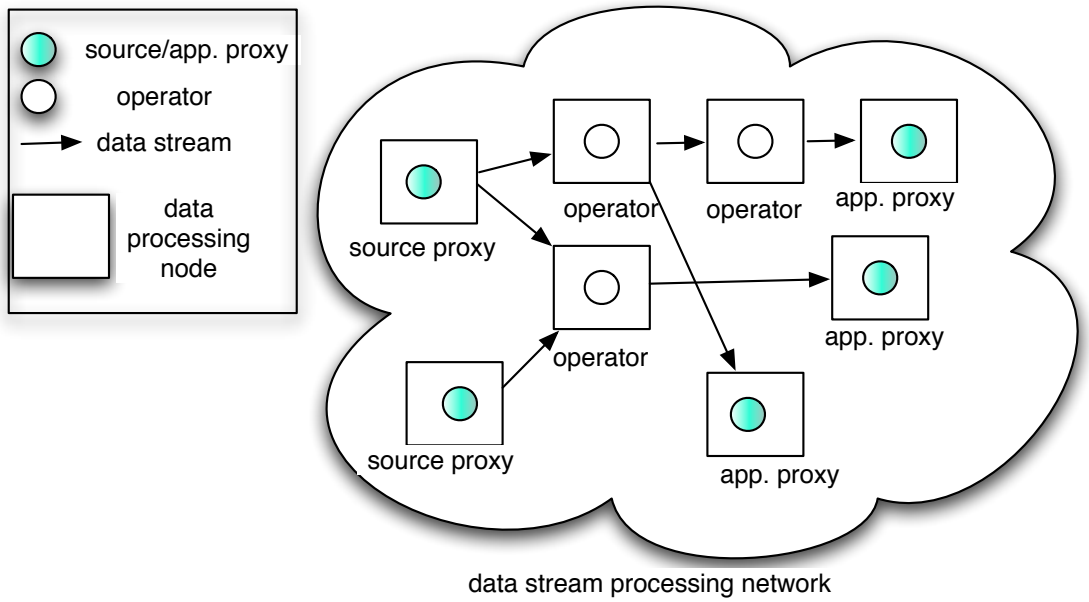


Figure 2.1: Deployment of in-network data-stream processing

downstream operator(s) or proxy (proxies) as shown in Figure 2.1. How the operator graph is generated and deployed in the network is an orthogonal problem to this dissertation. The system can generate and deploy operator graphs based on each application's subscription, or an application can explicitly specify an operator graph and its deployment [21, 22, 7].

Our bandwidth optimization focuses on data sources or operators that need to send data to remote downstream operators or proxies via multicast. We assume that data are used at different granularity by the downstream operators and that the data-quality requirements of its downstream operators have been communicated to the node that hosts the operator (see Figure 2.2). How the data quality requirements are derived for each operator from end-to-end application requirement, and how the requirements are propagated to each operator, are orthogonal to this dissertation.

We assume that the overlay infrastructure supports application-level multicast, and that the fan-out degree of each operator/proxy is of medium size, such that the overhead of managing multicast groups is moderate or negligible. The application-level multicast is

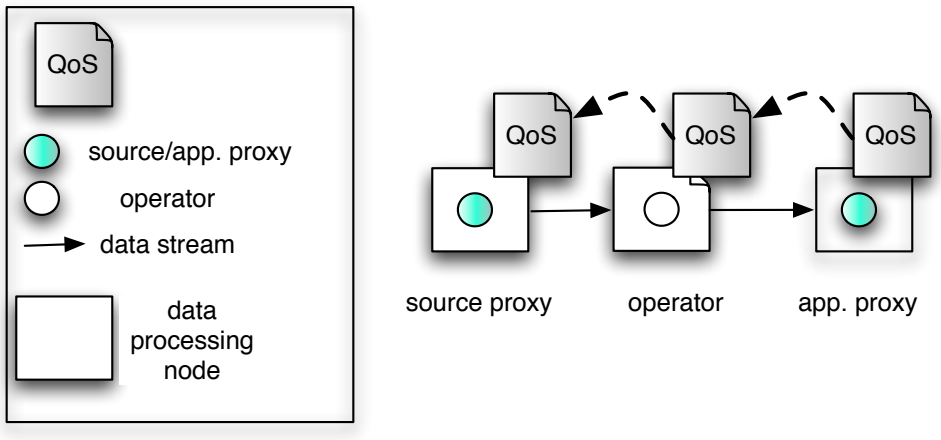


Figure 2.2: Data quality specifications propagates from applications to the sources

operator-oriented, in that data from one operator can be multicast to its downstream operators. We call each item in a stream a *tuple*. Further, we assume we use a multicast protocol that supports tuple-level data multicast, i.e., each tuple may or may not share the same multicast group. Such multicast protocols have been intensively studied in web objects dissemination systems to support differentiated data repository sharing [38].

We assume data sources are infinite and time-ordered series with self-describing data types. A tuple consists of a collection of attribute-value pairs. The value of an attribute can be as simple as an integer, or as complex as a web object that contains hierarchy of documents in text or images. Also, we assume that all tuples are timestamped at the originating sources.

2.2.2 Group-aware stream filters

The filters that fit for group-aware stream filtering usually exhibit the following properties.

- The filters are exclusively data-selection filters; that is, the output of a filter should be a subset of its input tuples.

- For each output, a group-aware filter can offer a set of candidate tuples that are equivalent in quality, such that the filter is equally satisfied with any of the candidate tuples.

- A group-aware filter has to choose all candidates of an output before choosing any candidates of its next output.

- A group-aware filter is able and obligated to finish choosing candidates of an output, if the system asks it to do so (for timeliness, as we show in Chapter 3).

- A group-aware filter computes the candidates for an output in an on-line fashion. It is still possible for a filter to adjust the set of candidates for an output before moving on to computing the set for the next output.

In the next section, we introduce a simple filter with these properties. In Chapter 5 we introduce other filters and consider their properties.

For group-aware filtering, an application needs to choose a filter function and specify its parameters, along with a latency-tolerance parameter. How applications pick the parameters to meet their domain-specific requirements is out of the scope of this thesis.

2.2.3 Reference-based candidate sets

Based on our key observation that there may exist more than one legitimate output that satisfies a filter's function specification, we introduce the notion of *candidate set* to capture the set that contains all equivalent outputs of a filter.

Given a data stream segment, there exist many domain-specific ways for a filter to compute its candidate set. First we introduce a *reference-based approach* to do that. Later in the chapter we introduce another way to compute candidate sets. The reference-based approach lets the filter compute a set that contains tuples equivalent to each tuple that the

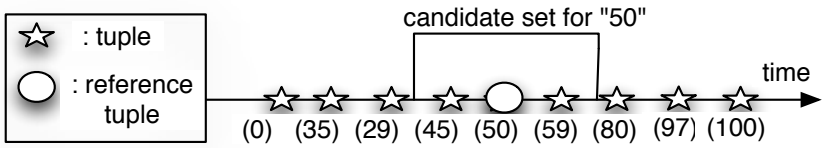


Figure 2.3: Candidate set of a reference tuple.

self-interested filter would select to output. We call the tuples that would be chosen by a self-interested filter *reference tuples*, and the set of candidate tuples for each reference tuple the "candidate set". Choosing any tuples from the candidate set corresponding to a reference tuple would be quality equivalent to choosing the corresponding reference tuple for the output. Figure 2.3 shows a Delta Compression (DC) filter that can tolerate 10-unit slack in 50-unit compression, and can select a vicinity of tuples around the reference tuple, here tuple 50, that are no more than 10 units from the reference tuple, to form its candidate set. In this case, tuples whose values are 45, 50, 59 are within the 10-unit vicinity of, and contiguous with, the reference tuple 50, and thus make the candidate set. We assume for now that every candidate set is finite or *closable*. In Chapter 3, we introduce mechanisms to handle the cases where candidate sets may be unbounded in length.

2.2.4 Problem definition

Before formally defining the group-aware stream filtering problem, we first assert important properties of candidate sets in our consideration.

Definition 1 *A time cover, TC_i, of a candidate set i is [min $\{\forall t_j | t_j$ is time stamp of tuple j in candidate set $i\}$, max $\{\forall t_j | t_j$ is time stamp of tuple j in candidate set $i\}$]*

Axiom 1 *Time covers of a group's candidate sets do not intersect.*

We assert this axiom to make sure that candidate tuples remain in temporal order; that is, if a reference tuple A's time stamp is smaller than reference tuple B's, we make sure the time stamp of any of the candidates for A is smaller than that for all the candidates for B. In a delta-compression filter, the axiom requires that the quality slack is less than half of the delta value, which is normally desirable.

The group-aware stream filtering problem for a finite data stream can be defined formally as follows.

Problem Definition 1 *Given a finite stream segment S, n filters $F_1, F_2, ..., F_n$ in the group, and a collection C containing all candidate sets produced by the filters. The objective is to pick a tuple o_j from each candidate set in C, such that the set $Output = \bigcup_j o_j$ has minimal size.*

We now prove the group-aware stream filtering problem is NP-hard.

Theorem 1 *Group-aware stream filtering is NP-hard.*

Proof: We prove this property by reducing the problem to the minimum hitting-set problem, which is a classical NP-hard problem [25].

Consider a special instance of the group-aware filtering problem in which each filter F_i has exactly one reference point and thus exactly one candidate set $cands_i$ for input stream S. Suppose we have n filters, so there are n candidate sets to choose output from. Since each candidate set is a subset of the tuples in S, this problem has a solution if and only if the minimum hitting set problem with these n sets has a solution, that is, the output O of the minimum hitting set problem makes sure that every one of the n sets intersects with (or "hits") O, and O's size is smallest among all solutions. □

The minimal hitting set problem has been studied extensively in the computer science literature. It is proved that the greedy algorithm produces a $\rho(n)$approximation to the optimal solution [25], where $\rho(n) = H(max\{|C| : C \text{ is a set in the hitting-set problem}\})$

and where H is a harmonic function. We can apply this bounded approximation algorithm directly for the group-aware filtering problem.

Notice that the problem we defined assumes that the input is a finite-length time series. For a continuous event stream that is potentially infinite in length, we consider a group-aware filtering optimization problem for all its finite prefixes of a time-ordered input data sequence.

Problem Definition 2 *Given an input stream S=$< D_0, D_1, \ldots >$, and a group of n filters $F_1, F_2, ..., F_n$, we consider a prefix segment S_t of S at any time point t, that is, all data that have reached the filters till time t. For the finite segment S_t, suppose C is a collection of all candidate sets produced by the filters. The objective of group-aware filtering is to pick a tuple o_j from each candidate set in C, such that the set $Output = \bigcup_j \{o_j\}$ has minimal size.*

2.3 Framework for group-aware stream filtering

In this section, we show the framework for group-aware stream filtering. First, we introduce the two-stage process at the core of the group-aware stream filtering problem. Next, we provide a scheme of segmenting the source data for applying hitting-set algorithms and show two properties of this scheme, based on which we show two heuristics-based algorithms for the problem.

2.3.1 Two-stage process

We abstract our group-aware stream filtering method into two stages (Figure 2.4). In the first stage, *compute candidate outputs*, each filter processes the input stream and computes a set of candidate outputs; in the second stage, *decide on final outputs*, an output decider

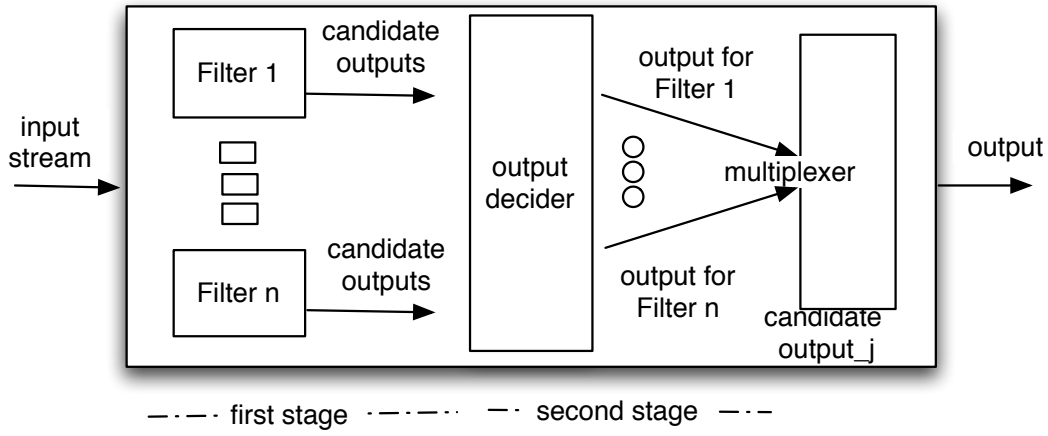


Figure 2.4: Two-stage process in group-aware stream filtering.

chooses a candidate output for each filter for multicasting. For a continuous stream, group-aware filtering iteratively goes through these two stages, processing one segment at a time.

2.3.2 Region-based segmentation

For a continuous source stream that is potentially infinite in length, it is impossible to gather all data before applying the greedy algorithm. Is there a way to segment the input time series in such a way that the segmentation does not affect the optimality of the solution? Here we propose a *region-based segmentation* so we may apply the greedy algorithm.

Definition 2 *If A and B are candidate sets from two filters, and the time covers of A and B intersect, we say A and B are "connected".*

Definition 3 *If A and B are connected candidate sets, and B and C are connected candidate sets, we deem that A and C are also connected, although the time covers of A and C may not intersect.*

Definition 4 *A region is a maximum family of candidate sets such that each set is connected with every other in the family.*

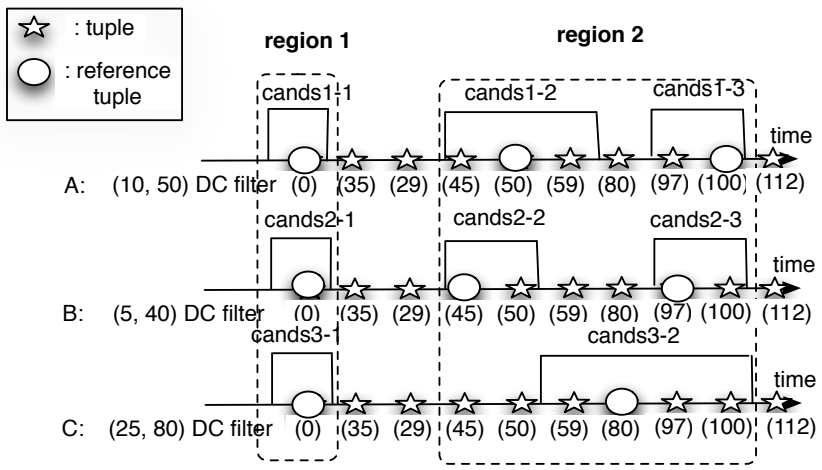


Figure 2.5: Two regions for three DC filters.

Definition 5 *A time cover for a region is the union of all time covers of the candidate sets contained in the region.*

Figure 2.5 shows that there are two regions based on three DC filters' candidate sets: $region_1 = \{$cands1-1, cands2-1, cands3-1$\}$, $region_2 = \{$cands1-2, cands2-2, cand3-2, cands1-3, cands2-3$\}$. It is easy to verify that adding any candidate set outside a region to the region will invalidate the region, as the added candidate set is not connected with the rest of the sets in the region.

Axiom 2 *Different regions' time covers do not intersect.*

Proof: we prove it by contradiction. Suppose the time covers of two regions, A and B, intersect. It is easy to see that at least one candidate set, say $cands_0$ in B, is connected with a candidate set in A. Then adding a new candidate set $cands_0$ to A will still make A a region, which directly contradicts the assumption that A is a maximum collection of the connected candidate sets. □

Theorem 2 *Given an input time sequence S, applying divide-and-conquer of the group-aware filtering to each region in S will not affect the optimality of the solution.*

Proof: We need to prove that a set-union of the optimal solutions from each region on the input stream S is an optimal solution for S. We prove it by contradiction: that is, we suppose the opposite is true: given a total of n regions on S, and each region has an optimal solution O_i, the cardinality of the optimal solution O' of S is smaller than the size of the set-union U of all O_i, thus U is not an optimal solution for S. Now we divide O' into n distinctive subsets such that each subset is a group-aware filtering solution on each region, that is, each subset is a hitting set of a region.

To find n such subsets, we can first initialize n empty auxiliary sets, one for each region; then, for each tuple in O' that is contained by one of the candidate sets in a region, we put it in the auxiliary set of that region. We can see each tuple fall into exactly one auxiliary set; otherwise if a tuple belongs to two auxiliary sets, then there must be two candidate sets from two different regions containing the tuple, which means that the two candidate sets are connected and thus belong to the same region, which contradicts the assumption that they are from two different regions. In the end, we get n distinct subsets of O' in the auxiliary sets. We can prove that each auxiliary set is a hitting-set solution to its corresponding region. We prove it by contradiction. Suppose the opposite is true: that is, at least one candidate set in a region does not intersect with ("hit") the auxiliary set corresponding to the region. We know none of the other auxiliary sets hit this candidate set, otherwise there must be a candidate set in another region that intersects with this candidate set, which means that they are connected and are in the same region, which reaches a contradiction to our assumption.

As the size of O' is smaller than that of U, there must be at least one of the n subsets of O' whose size is smaller than that of the optimal solution O_j of that region, which

contradicts the optimality of O_j for the region. □

2.3.3 Heuristics-based algorithms

Since the group-aware filtering problem is NP-hard, it is desirable to use heuristics-based algorithms to solve problem approximately.

Region-based segmentation not only preserves the optimality of an optimal solution on the group-aware filtering problem, but also preserves the maximum approximation ratio of a heuristics-based algorithm.

For heuristics-based algorithms that find sub-optimal solutions for the group-aware filtering problem, we prove that region-based segmentation preserves the approximation ratio of the solution.

Theorem 3 *Region-based segmentation preserves the maximum approximation ratio of a heuristics-based algorithm. That is, given an input source segment S, which is segmented into n regions, if a heuristics-based algorithm has approximate ratio r_1, r_2,..., r_n on each region respectively, the approximation ratio r of the algorithm on the overall segment S satisfies the property that $r \leq \max(r_1, r_2, \cdots, r_n)$.*

Proof: Suppose $r_i = \max(r_1, r_2, \ldots, r_n), 1 \leq i \leq n$, that is, the approximate ratio of the algorithm in the ith region is the biggest among approximate ratios of all regions. According to the definition of approximation ratio, $r_j = \frac{O'_j}{O_j}, 1 \leq j \leq n$, where O_j is the size of the optimal solution on the jth region, and O'_j is the size of the sub-optimal solution produced by the heuristics-based algorithm on the jth region. According to the assumption that r_i is the bigger than the approximate ratio of any other region, we get $O'_j \leq \frac{O_j * O'_i}{O_i}$, where $j \neq i$. The approximation ratio r on the overall data segment S satisfies

$$r = \frac{O'_1 + O'_2 + \cdots + O'_n}{O_1 + O_2 + \cdots + O_n}$$

$$
\begin{aligned}
&\leq \frac{\frac{O_1 * O'_i}{O_i} + \frac{O_2 * O'_i}{O_i} + \cdots + \frac{O_n * O'_i}{O_i}}{O_1 + O_2 + \cdots + O_n} \\
&= \frac{\frac{(O_1 + O_2 + \cdots + O_n) * O'_i}{O_i}}{O_1 + O_2 + \cdots + O_n} \\
&= \frac{O'_i}{O_i} \\
&= r_i \\
&= \max(r_1, r_2, \ldots, r_n) \qquad \square
\end{aligned}
$$

Region-based greedy algorithm

Now we introduce a region-based greedy algorithm REGION-BASED-GREEDY-FILTERING for a continuous stream S in Figure 2.6. First, assume that we have instantiated each filter according to its specification from each application. A filter specification specifies the type and parameters of the filter, and how its internal state should be initiated and updated. We use a global object $globalState$ to coordinate the filtering. The global state mainly consists of 1) the *group utility* of each tuple, which counts the number of filters that have included the tuple in their candidate set, and 2) the current *region* that keeps track of the connected candidate sets since the last region. Each filter uses its $isAdmissible$ (line 3) method to decide whether a tuple is admissible to its candidate set. If so, the tuple is added to the filter's candidate set (line 5), and the tuple's group utility is incremented in the $globalState$ (line 7). A filter's $isAdmissible$ method may tentatively admit tuples based on estimates of the next reference tuple. When a tuple is admitted and also marked as a reference output, the filter will need to check and dismiss candidates that are more than "slack" away from the reference output. (This means that some tuples initially considered as candidates are removed from the candidate set.) Next, if the filter finishes computing the current candidate set (line 8), then we use the $globalState$ to check if all connected

```
REGION-BASED-GREEDY-FILTERING(S)
 1  while ( (currentTuple ← S.getNextTuple()) ! = null)
 2      do for each filter f in the group
 3           do if f.isAdmissible(currentTuple)
 4                then ▷ first stage: admit candidates
 5                     f.candidateSet.add(currentTuple)
 6                     f.state.update(currentTuple)
                       ▷ increment group utility of currentTuple
 7                     globalState.groupUtility.increment(currentTuple)
 8              if f.candidateSet.closed(currentTuple)
 9                then globalState.addClosedCandidateSet(f.candidateSet)
          ▷ second stage: if current region is ready, decide output based on globalState
10        if (region ← globalState.getCurrentRegion()) ! = null)
11          then ▷ apply greedy algorithm to the region to decide the output
12               output ← GREEDY-HITTING-SET(region, globalState.groupUtility)
                 ▷ multicast output
13               multicaster.send(output)
```

Figure 2.6: Region-based greedy algorithm for group-aware stream filtering.

candidate sets are closed. If the utility of any tuple in a closed candidate set is greater than the number of currently closed candidate sets, then the region is not closed, as there must be a not-yet-closed candidate set admitting this tuple (line 10). When the current region is closed, it consists of all the closed candidate sets that are connected. Next, we apply a greedy hitting-set algorithm, GREEDY-HITTING-SET (Figure 2.7), to the current region (line 12) and send the solution for multicast (line 13). The solution contains a set of tuples chosen from the region that have high group utilities and that hits all candidate sets in the region.

GREEDY-HITTING-SET (Figure 2.7) picks the tuple with the highest group utility (line 3). If multiple tuples have the same highest utility, we use tuples' time stamps to break the ties and choose the tuple with the latest time stamp to favor time freshness. Then, we remove all the candidate sets that contain the chosen tuple (line 5). The group utility of any tuple

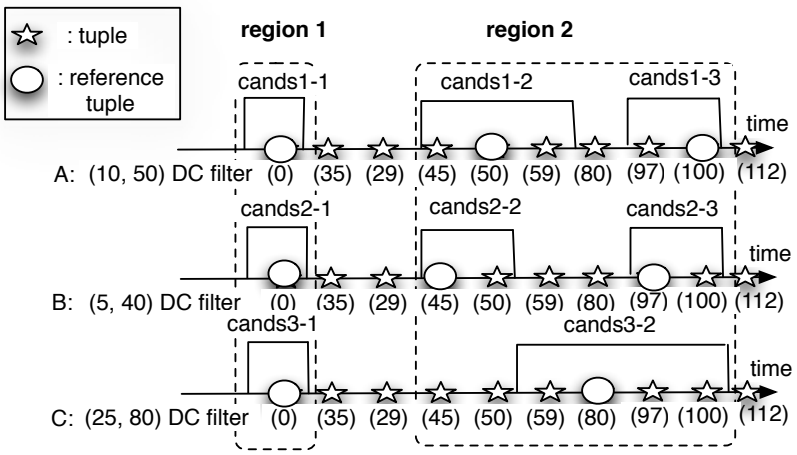
```
GREEDY-HITTING-SET(region, groupUtility)
1  resultSet.init(∅)
2  while (region.hasMoreCandidateSets())
   ▷ greedily pick the tuple with max groupUtility; use time stamp to break ties, if there are any
3      do maxUtilityTuple ← getMax(groupUtility)
4         resultSet.add(maxUtilityTuple)
          ▷ get all the candidate sets that are hit by maxUtilityTuple
5         hitCandidateSets ← region.removeAllCandidateSetsHitBy(maxUtilityTuple)
          ▷ decrement groupUtility for each tuple in the hitCandidateSets
6         groupUtility.decrUtilityFor(hitCandidateSets)
7  return resultSet
```

Figure 2.7: Greedy hitting-set algorithm.

included in the removed candidate sets is decremented by the number of removed candidate sets containing the tuple (line 6). The algorithm then greedily picks the next tuple with the highest utility and the same hitting-set process continues until no candidate set is left to be hit. The chosen tuples constitute the solution.

Figure 2.8 shows an example of how the region-based greedy algorithm is applied to three group-aware filters A, B and C. The upper part of the figure shows the candidate sets of the filters. The lower part shows the time-progressive computation of group utilities of the tuples, status of regions, and chosen outputs for the group.

At each time slot, each filter checks whether the newly arrived tuple is admissible. If so, the filter increments group utility of the tuple. At time slot 2, all three filters' candidate sets are closed, thus the region is closed to run greedy hitting-set algorithm. The output 0 is chosen for all filters. The next region closes at time slot 10, when all filters close their candidate sets. Tuple 100 is chosen as it is one of the tuples with the highest group utility. That is, cands1-3, cands2-3, cands3-2 are "hit" by tuple 100. Next, tuple 50 is chosen, as it has the next highest group utility. cands1-2 and cands2-2 are both hit by tuple 50. Now, all candidate sets have been hit in region 2. Thus, the outputs chosen at time slot 10 are tuple



time	1	2	3	4	5	6	7	8	9	10
tuple	0	35	29	45	50	59	80	97	100	112
group utility	3	0	0	2	2	2	1	3	3	0
region status		closed								closed
chosen outputs		0->{A, B, C}								100-> {A,B,C} 50-> {A,B}

Figure 2.8: Region-based greedy algorithm for three DC filters.

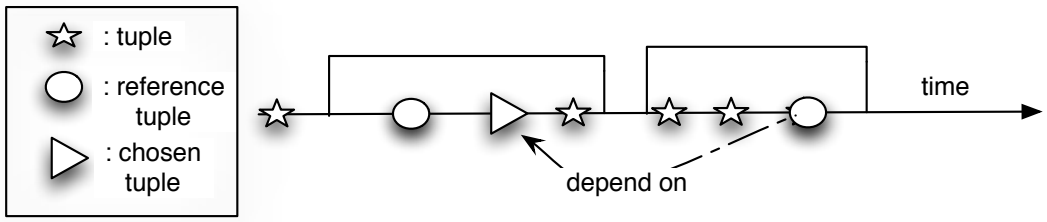


Figure 2.9: Stateful candidate sets.

100 for filter A, B and C, and tuple 50 for filter A and B.

Per-candidate-set greedy algorithm

Notice that, so far, we assume that candidate sets are computed with the reference-based approach, that is, computing a filter's current candidate set does not depend on the chosen output of its previous candidate set. We call this *stateless computation of candidate sets* for a filter. For some applications, an alternative semantics requires a candidate set to base its reference on the tuple chosen for output from the previous candidate set. We call it *stateful computation of candidate sets*, and call the candidate sets *stateful candidate sets*.

For stateful candidate sets, the filter needs to choose the output as soon as its current candidate set closes, as the reference for the next candidate set depends on the chosen output (Figure 2.9). We use group state to track already-chosen tuples of each stateful candidate set in addition to the group utilities of tuples, and propose the following two heuristics for choosing output tuples from stateful candidate sets:

- choose the tuple that has been chosen by other filters.
- choose the tuple that has the highest group utility.

The first heuristic takes precedence over the second heuristic. Both are subject to the tie-breaking rule, preferring the more recent tuple. After a filter chooses a tuple from a candidate set, the group utilities of all tuples in its candidate set are decremented by 1. Group state keeps track of the tuples chosen by each filter. If there are stateless filters in the group, computing regions is still useful, as it is the earliest possible time to multicast chosen tuples that have not yet been output in the region, when a region closes. In that case, we can still apply the greedy hitting-set algorithm at the time the region is closed, only the stateful filters' candidate sets become singleton sets with one chosen tuple in each.

The essence of the per-candidate-set greedy algorithm is to allow a filter to choose by itself the output once its current candidate set is closed, rather than waiting for all its connected candidate sets to close as it does in region-based greedy algorithm. In this sense, per-candidate-set is also applicable for stateless candidate sets.

PER-CANDIDATE-SET-GREEDY-FILTERING (Figure 3.3) shows the general process of the per-candidate-set based greedy algorithm for group-aware filtering. At the first stage, each filter admits tuples from the source stream S (line 3) and updates $groupUtility$ of $globalState$ (line 7). If the current candidate set is closed, the filter, at the second stage, decides its output tuples for the current candidate set (line 9), based on the two rules we described above. For the decision, the filter consults both the $groupUtility$ of $globalState$, and outputs already decided by other filters stored in $decidedOutput$ of $globalState$. The latter is checked by the filter first. The output of the filter is then added to $decidedOutput$ of $globalState$ (line 10). After each filter finishes processing $currentTuple$, $multicaster$ multicasts and then clears up the decided output stored in $decidedOutput$ of $globalState$ (line 11).

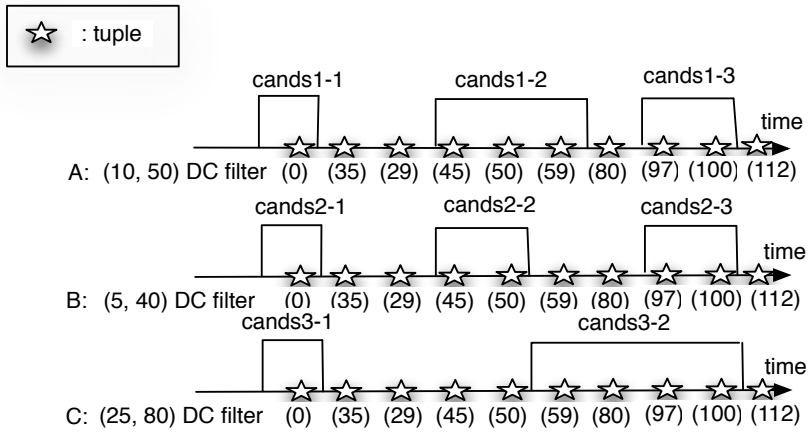
Figure 2.11 shows an example of how the per-candidate-set-based greedy algorithm is applied to the same three group-aware filters A, B and C of the previous example. The lower part of the figure shows time-progressive computation of group utilities of the data,

```
PER-CANDIDATE-SET-GREEDY-FILTERING(S)
 1  while ( (currentTuple ← S.getNextTuple()) ! = null)
 2      do for each filter f in the group
 3           do if f.isAdmissible(currentTuple)
 4                then ▷ first stage: admit candidates
 5                     f.candidateSet.add(currentTuple)
 6                     f.state.update(currentTuple)
                       ▷ increment group utility of currentTuple
 7                     globalState.groupUtility.increment(currentTuple)
                  ▷ second stage: if current candidate set is closed, each filter decides its output
 8                if f.candidateSet.closed(currentTuple)
 9                  then output ← f.decideOutput(globalState)
10                       globalState.decidedOutput.add(f, output)
          ▷ multicast output multicasts and clears up the decided output
11        multicaster.send(globalState.decidedOutput)
```

Figure 2.10: Per-candidate-set based greedy algorithm for group-aware stream filtering.

of which filter's candidate set is closed at each time slot, and of outputs chosen by each filter. At time slot 2, all three filters' candidate sets are closed, thus each needs to choose output, and naturally tuple 0 is chosen for all filters, as that is the only tuple contained in all three candidate sets. At time slot 6, filter B's candidate set is closed and B thus needs to choose an output. Since both tuples in its candidate set have the same group utility and tuple 50 has a fresher timestamp, B chooses the tuple 50 as the output, as the second heuristic applies. The information about B's chosen outputs is communicated to the global state shared with the other two filters. When at time slot 7 when filter A's candidate set closes, it checks the first rule for choosing outputs, that is, choose the tuple(s) already decided by other filters. Since tuple 50 is already chosen by filter B, A goes along and chooses tuple 50 as its output. At time slot 10, all three filters' candidate sets close. According to the second heuristic subject to the tie-breaking rule, A chooses tuple 100 as the output. Then B and C follow along and choose tuple 100 as their output as well, as the first heuristic dictates.

Notice that given a prefix sequence of a data source, the region-based group-aware fil-



time	1	2	3	4	5	6	7	8	9	10
tuple	0	35	29	45	50	59	80	97	100	112
group utility	3	0	0	2	2	2	1	3	3	0
closed cand. sets		A, B, C				B	A			A,B,C
chosen outputs		0->{A, B, C}				50->{B}	50->{A}			100-> {A,B,C}

Figure 2.11: Per-candidate-set-based greedy algorithm for three DC filters.

tering algorithm does not affect a filter's data compression ratio, because for each reference output, there is a corresponding tuple chosen by the group-aware filter, and vice versa. In the case of the per-candidate-set algorithm, this may not be true, as the candidate sets are not based on reference outputs of self-interested filtering and thus there is no one-to-one correspondence between reference outputs of self-interested filtering and candidate sets of per-candidate-set group-aware filtering. Decreased compression ratio in a group-aware filter is not desirable in terms of bandwidth savings. One potential solution is to compute the value-changing trend and admit only those candidates whose value updates match the value-changing trend. We may also assume there is an equal chance for a per-candidate-set group-aware filter to have an enlarged or shrunken compression ratio. For long-running data sources, we can assume the overall compression ratio is not affected.

2.4 Related work

We detail related work in the following four areas: imprecise processing, group-oriented optimization, fine-grain multicasting, and alternate heuristics-based algorithms applicable to the group-aware filtering problem.

2.4.1 Imprecise processing

Imprecise processing is not a new concept in computer science. As early as about twenty years ago, the concept was first introduced to real-time systems to deal with transient overloads to improve system availability [43, 44]. It hinges on the fact that intermediate, approximate results with a poorer but acceptable quality on a timely basis are preferred to late results of the desired quality. Imprecise computations for real-time systems have multi-version modules that create flexibility in task scheduling. In times of overload, it helps to degrade the system performance gracefully and to enhance system fault tolerance and

availability. Imprecise computation is also related to the general term of "anytime computation" in AI literature [75]. Anytime computation introduces a trade-off between run-time and quality of results. But the focuses in AI are the algorithmic aspects and the reasoning about the results produced by the algorithms.

The stream-processing systems of recent years support monitoring applications that often have soft deadlines [1] or are able to tolerate some delay. Once again, the systems embrace imprecise processing as a core methodology to alleviate computation pressure of unending data streams on "blocking" data operations that typically requires consuming all input before producing precise output. The imprecise processing boosts data availability, and significantly reduces the computational cost for complex data operations, such as join operations. One natural technique for imprecise processing of data streams is to use sliding windows to compute results only on the most recent portion of data. STREAM [2, 3] extends SQL and relational algebra to explicit window specifications to form the Continuous Query Language (CQL) for querying continuous data streams. Window-based join operation has attracted a lot of research work from the database community. How the window affects the quality of the results is still a challenge. Another kind of imprecise processing techniques for streams involves sampling or batch processing of data, rather than processing every data element as it arrives. When the incoming data rate is slower than the processing speed, the natural solution is to process data in batches. The result is approximate as it is not timely: it represents the exact answer at a point in the recent past rather than the current-moment answer. For cases when the incoming data rate is faster than the processing speed, data are evaluated over a sample of the data stream rather than over the entire data stream. For example, Aurora [7] introduces random-sampling-based data shedding operators that can be dynamically inserted into the work flow to reduce the data load. Much of the research effort in the past several years has been devoted to deriving confidence bounds on the error introduced by the sampling process, and to designing sampling-based algorithms

that produce approximate answers that are provably close to the exact answer [6, 9, 66]. Yet another imprecise-processing technique is to compute synopses or digests that maintain a small amount of streaming data to keep computation per new data element to a minimum. Common techniques to generate synopsis structures include sampling [6, 9, 20, 33, 49], sketching [5], building histograms [31], and wavelets [18].

Johnson et al. [33] summarized a general structure for sampling operators for queries expressed in CQL. The structure also contains candidate-set admitting and output-deciding stages, as we propose for the general group-aware filtering process. If we see the group-aware filtering from a sampling point of view, our algorithm is a special kind of sampler in that it picks an output from a candidate set of outputs for each filter. But our process involves coordination across a group of applications, which never occurs in Johnson's single-application sampling.

Our group-aware filtering algorithms also produce approximate results in that they use batch processing based on regions for the region-based greedy algorithm, or based on candidate sets for the per-candidate-set greedy algorithm, and the result is not exactly timely. And the quality of results of the heuristics-based algorithms may not be optimal. The key difference between the group-aware filtering and data operations for approximate stream processing discussed elsewhere lies in the trade-offs we make. We trade off computation time for reduced data load to save network bandwidth; while other existing data-stream oriented operators trade quality for reducing computation time or to increase the throughput of the system. The distinct difference in trade-offs is dictated by the constraints a system faces. We consider network bandwidth to be the most constrained resource, so we are willing to spend CPU time to save network bandwidth; while most other stream-processing optimization systems focus on reducing CPU and memory costs for stream processing, as they assume plentiful network bandwidth.

Exploit application semantics. Our work exploits the semantics of a stream processing application to improve resource management in a distributed dissemination system. IBM's Gryphon [63] also leverages the semantics of subscribing applications to compress a sequence of data updates that have the same effect on applications' ultimate states. Zhao et al. [73] propose a special rule-based language to specify an application's sophisticated processing needs, specifically the semantical equivalence of outputs to a remote application in the face of retransmitted and disordered data. Rather than using a complicated language to describe the needs, we opt for a simple framework that lets applications describe their filtering requirements with filter types and parameters.

2.4.2 Group-oriented optimization

Many stream-processing techniques are discussed in the context of a single streaming application. To ensure scalability, it is important to use shared execution of multiple stream applications. NiagraCQ [23] and Olston [52] consider shared execution of common sub-expressions of the predicates of multiple queries by using traditional compiler rewriting techniques. Madden [46] considers execution of multiple filters by using a novel sub-expression indexing structure. Multi-query optimization for continuous queries with "group by" features resorts to shared scans, shared sorts and shared partitions for efficient execution [4, 24]. The objective of all these group-oriented optimization methods is to take advantage of the commonality in data processing to reduce CPU cost. Our group-aware filtering is an optimization technique for a group of data-stream applications, only we have different objective: our goal is to trade computation time for reduced communication cost.

2.4.3 Multicasting

Our work uses filtering, in conjunction with application-level multicasting, to provide fine-grained multicast such that each data element in the source can potentially have a different multicast group. Such a type of multicast services has been used in multicasting web objects from a web repository [38], in that each object from the repository may go to different groups of clients. The overhead on group-membership management for this fine-grained multicasting is larger than that for multicasting the whole data set to a single group, or than that for IP multicast. The control overhead for the multicasting for a median-sized group, according to a comparative study with IP multicast [39], can be acceptable to exploratory monitoring applications and the cost can be amortized over a long period if the group changes infrequently. Application-level multicast infrastructures [17, 58, 55, 67, 74], such as CAN and Pastry, create and manage multicast groups and build efficient multicast forwarding trees.

In the area of multimedia data dissemination, filters can be installed along a multicast tree to provide quality-differentiated multimedia streams to clients with disparate capabilities and QoS requirements [71, 32], a high-level goal similar to our work. The filtering mechanisms that enforce quality-aware sharing of multimedia data use many types of filters such as selectors, transformers and mixers that are specifically designed for processing continuous multimedia data, while the filters used in our work are exclusively for tuple selection. Filter placement in a multicast environment is a key engineering challenge, valuable both for event-based stream-processing systems like ours, and for multimedia streaming systems. Our dissertation focuses on how to make tuple-selection filters group-aware after being strategically deployed in the network by some other mechanism. Finally, quality grading of multimedia data is dependent on human perception capability. For event data streams data quality is up to each application's semantics— and these semantics vary

widely from application to application.

2.4.4 Heuristics-based Algorithms

Beyond using the greedy algorithm for solving the minimal hitting set problem at the core of the group-aware filtering problem, other popular heuristics-based algorithms include simulated annealing, neural networks, and genetic algorithms [60].

Simulated annealing (SA) is a probabilistic algorithm for the global optimization problem, namely locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete, for example, all tours that visit a given set of cities. Inspired by the annealing process in metallurgy, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter (the temperature), that is gradually decreased during the process.

Neural networks, inspired by the architecture of the human brain, are weighted digraphs with neurons as vertices and weights on edges denoting the connection strength of the pair. Its practical use comes with algorithms designed to alter the weights of the connections in the network to produce a desired signal flow. Neural networks have been more successful in classification and forecasting applications than for combinatorial optimization problems, such as the one that our group-aware filtering dictates.

Genetic algorithms are implemented as a computer simulation in which a population of abstract representations (called chromosomes or the genotype or the genome) of candidate solutions (called individuals, creatures, or phenotypes) of an optimization problem evolve toward better solutions. A typical genetic algorithm requires two things to be defined: a genetic representation of the solution domain, and a fitness function to evaluate the solution domain. Genetic algorithms maintain a "population" of solution candidates for a given

problem. Elements are drawn at random from this population and allowed to reproduce, by combining some aspects of the two parent solutions. The probability that an element is chosen to reproduce is based on its fitness, essentially a function of the cost of the solution it represents. Eventually, unfit elements die from the population, to be replaced by successful-solution offspring. Although the idea is extremely appealing, it does not seem to work as well on practical combination optimization problems as simulated annealing, as it is quite unnatural to model most applications in terms of genetic operators like mutation and crossover. Also, it takes a long time to compute on nontrivial problems.

All those complex evolutionary algorithms take much longer to find a good solution given a non-trivial problem, compared with a deterministic greedy algorithm. For timeliness concerns, we opt out of these types of algorithms in solving the group-aware filtering problem.

Chapter 3

Enforce Data Timeliness

In this chapter we focus on methodologies to enforce data timeliness in group-aware stream filtering. First, we introduce the requirement and the models for data timeliness. Then we illustrate two timeliness-enforcing strategies used for group-aware filtering: timely cuts and output strategy. Finally, we discuss related work with particular focus on the methodologies used in real-time and other data stream systems to satisfy applications' time requirements.

3.1 Timeliness requirement

Distributed stream-processing systems help monitoring applications to acquire desirable data and thus the systems are obligated to satisfy data-quality requirements of the applications. Unlike real-time applications, many monitoring applications are exploratory in nature and thus the requirements are "soft" and act as a guideline to the stream-processing systems to provide best-possible services. Also, applications may be aware of unpredictable dynamics in system resources, such as the bandwidth change of a wireless network in a harsh environment, and are willing to adapt their data requirements according to system conditions. For instance, consider a robot-location tracking application that normally wants

location updates at a data rate of 1 second per reading. In times of severe network conditions, such as when the network bandwidth drops to less than 500kbps, it may be willing to degrade requirements for location updates to a slower rate, say 5 seconds per reading.

The data-quality requirements of an application need to be broken down for each in-network data-processing operator. Each operator knows about the data-quality requirements of all its downstream operators. In our work, we assume such a requirement engineering and propagation process has already taken place, and that at each in-network processing node, it keeps track of all data quality requirements for each operator from its downstream operators. For an operator shared by multiple remote downstream operators that have different data-quality requirements, the hosting node deploys a group-aware filter to provide quality-differentiated service to each of the subscribing operators. Details of the propagation are shown in Figure 3.1. Data timeliness requirements may also tangle with data selection requirements based on timestamps of the data. How to help applications to specify multi-dimensional quality requirements that are congruent is non-trivial, and is out of the scope of this thesis. We focus our effort on enforcing satisfiable timeliness requirements.

Data quality generally considers four aspects of quality: data accuracy, granularity, timeliness, and completeness. We assume that a filter always satisfies the data-accuracy requirement in that it does not change the value of selected data. Data granularity and completeness is dictated by the semantics of the downstream operator of the filter, and concerns the subset of the data that is selected in relation to the whole data input. It directly affects network bandwidth usage for transporting the output. The preceding chapter illustrates the data-selection process in a group-aware filter to enforce data granularity and completeness. The timeliness requirements for group-aware filtering specify guidelines on the maximum time delay it causes for each tuple in the data stream.

In this chapter we focus on strategies to enforce data timeliness in group-aware stream

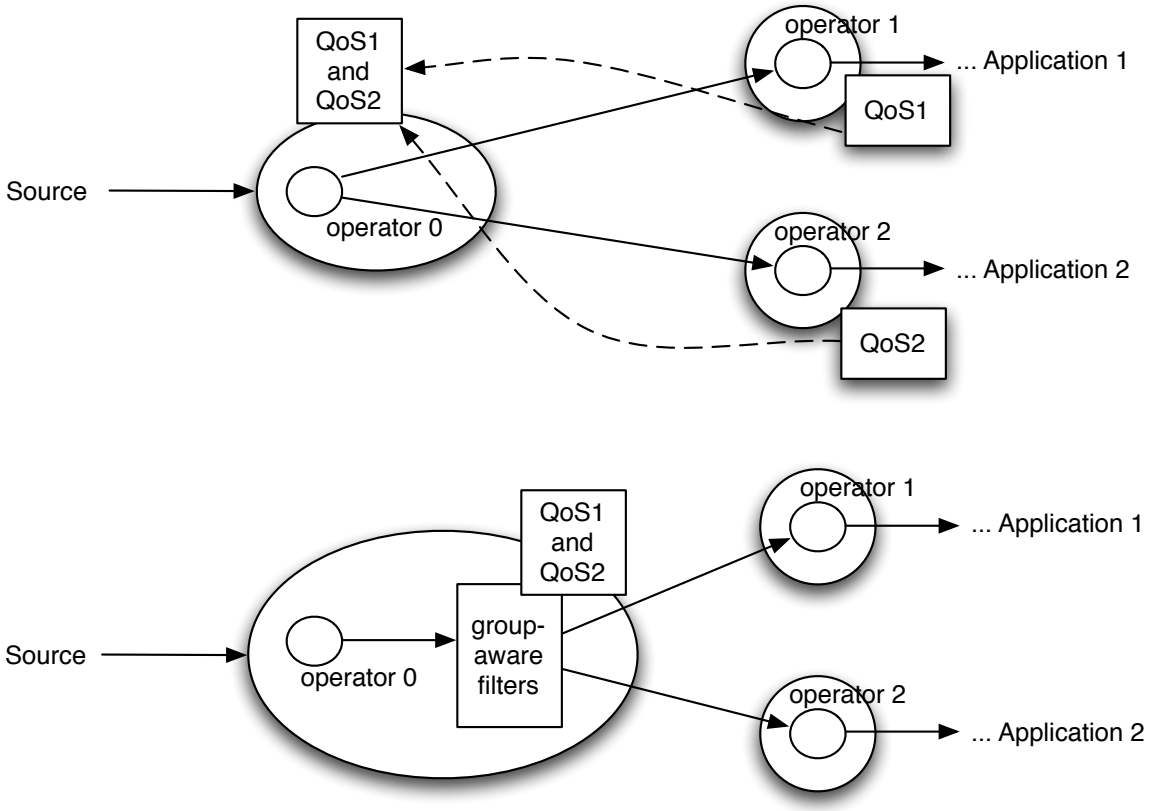


Figure 3.1: Group-aware filter with propagated group data-quality requirements

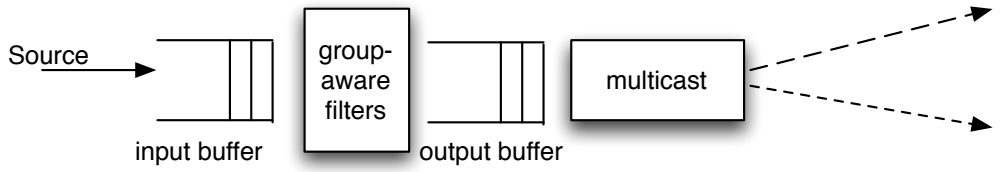


Figure 3.2: Data timeliness model

filtering. Group-aware filtering is based on batch processing of streaming data, so timeliness is first affected by the size of the batch. Secondly, the complexity of different algorithms leads to CPU cost, which affects timeliness. Finally, the output scheduling affects a tuple's stays in the output buffer of the filter before being multicast.

3.2 Timeliness model

Our timeliness model is concerned with the delay caused by group-aware filtering to the source data. Figure 3.2 illustrates four ways the timeliness of a tuple of the source data may be affected by the group-aware filtering process: 1) it waits in the input buffer of the group-aware filtering operator; 2) it is processed by the filter; 3) it waits in the output buffer of the operator; 4) it is multicast to the downstream operators. Thus, the delay caused by group-aware filtering process $D = D_{\textit{input buffer}} + D_{\textit{filter}} + D_{\textit{output buffer}} + D_{\textit{multicast}}$.

Note that the multicast delay $D_{\textit{multicast}}$ includes the delay in multiplexing data and invoking application-level multicast as well as the delay for transmitting and propagating the data over the network to reach the downstream operators or applications. The propagation and transmission delay is negligible compared with the delay caused by using application-level multicasting. For example, the propagation and transmission delay is about 1ms for 1M data over a 1Mbps network link, versus more than 50ms for invoking application-level multicast measured in a small simulated network [41, 42]. Also, multicast cost is largely dependent on the software implementation of the multicasting service, varying only slightly

by the amount of the data load.

With our simplifications made above, the timeliness model is primarily determined by D_{filter}, i.e., how fast the group-aware filters process the data, which in turn affects how long a tuple waits in the input buffer ($D_{input\ buffer}$). $D_{input\ buffer}$ is also affected by the incoming data rate. The bottom-line requirement for group-aware filtering is that its processing rate, compared with incoming data rate, should be fast enough not to cause congestion in the input queue. $D_{output\ buffer}$ is determined by the schedule for multicasting the output.

3.2.1 Model for region-based greedy algorithm

The region-based group-aware algorithm contains two stages. First, it computes regions, the smallest input stream segments, to which to apply the hitting-set algorithm, so that the optimality of the solution will not be affected. Then, when it closes a region, it applies the greedy algorithm for the hitting-set problem. Thus D_{filter}=$D_{accumulate\ region}$ + $D_{region\ greedy}$, where $D_{accumulate\ region}$ is the delay at the first stage and $D_{region\ greedy}$ is the delay at the second stage. It is easy to see that $D_{accumulate\ region}$ depends on the incoming data rate, assuming there is no congestion in the input buffer, and on the size of the region.

3.2.2 Model for per-candidate-set based greedy algorithm

The per-candidate-set based greedy algorithm differs from the region-based greedy algorithm in that the filter decides its outputs as soon as any candidate set closes. Thus $D_{pcs\ filter} = D_{accumulate\ cds} + D_{cds\ greedy}$, where $D_{accumulate\ cds}$ refers to the delay caused by accumulating a candidate set and $D_{cds\ greedy}$ to the delay caused by running the greedy algorithm for deciding outputs.

3.3 Timely cuts

The batch size of the data in a candidate set can be long, and potentially infinite. For example, a delta-compression filter, after admitting a tuple in the candidate set, waits for the first tuple that does not fall into the valid range for this candidate set to close the current candidate set. If the value changes little, and the filter's quality slack is relatively large, the candidate set can grow long.

Long candidate sets affect timely producing output in group-aware stream filtering, as it affects both terms in $D_{\textit{region filter}}$ or $D_{\textit{pcs filter}}$. For the region-based algorithm, the first term of $D_{\textit{region filter}}$, $D_{\textit{accumulate region}}$, refers to the time to accumulate the whole region. Thus computing long candidate sets adversely affects when a region closes. The second term of $D_{\textit{region filter}}$, $D_{\textit{region greedy}}$, is the delay caused by running the algorithm on a closed region. It is easy to see that the longer a region is, the longer it takes to compute the solution. Similarly, long candidate sets adversely affects $D_{\textit{accumulate cds}}$ and $D_{\textit{cds greedy}}$.

Hence, to curb the unfavorable effect of long candidate sets on satisfying timeliness requirements, we introduce a mechanism, *cuts*, to enforce time requirements for group-aware filtering. Cuts enforce early closure of candidate sets to avoid violating time constraints of the filters.

Predict algorithm's run-time Given a maximum allowed delay in a tuple $D_{\textit{region filter}}$, we need to predict the algorithm's run-time $D_{\textit{region greedy}}$ to decide how much time to allocate for computing the region for region-based greedy algorithm.

Suppose when the nth tuple is processed, the elapsed time t is less than $D_{\textit{region filter}}$. We need to predict whether admitting the next tuple will violate the time constraint. We need to predict the algorithm's run-time $D_{\textit{region greedy}}$ for $n{+}1$ tuples. If $D_{\textit{region greedy}}+$

t already violates the time constraint, we "cut" and close the region at the nth tuple. Otherwise, we will wait to process the next tuple for no longer than $D_{\textit{region filter}} - t - D_{\textit{region greedy}}$.

For predicting the region-based greedy algorithm's run-time, we build a latency model based on on-line observations of the most recent, say ten, regions' performance, specifically the correlation between region sizes and the CPU time for running the greedy algorithm. From our experiments we found that a linear model was a reasonably accurate fit. Thus, the computing time for n-tuple region will be about $n * slope + intercept$, where the $slope$ and $intercept$ can be calculated from a linear regression on recent run times. To be more conservative in meeting the timeliness requirements, group-aware filtering may apply overestimation to the run-time with an added constant to the predicted value.

Although predicting the running time for the per-candidate-set based algorithm can also resort to online monitoring and model fitting, we simply enforce a sorted data structure for maintaining the group utilities so that finding the tuples with highest group utilities only takes a small and constant time. We do not predict the computing time of the per-candidate-set based greedy algorithm.

REGION-BASED-GREEDY-FILTERING-WITH-CUTS in Figure 3.3 shows the process of enforcing timely cuts to region-based greedy filtering. We check the time constraint after each filter finishes processing the new input tuple (line 8). Then, if the time constraints are violated, we force all open candidate sets to close. These closures make the current region close automatically and then we can apply the GREEDY-HITTING-SET algorithm to choose the output, as before. Finally, after line 13, we let the $globalState$, which keeps track of CPU time for running the greedy algorithm on the region, update the algorithm run-time model, which will be used in the next region.

Now, let us go through the process of applying the region-based greedy algorithm with a timely cut to three group-aware filters, as in the previous examples; see Figure 3.4. At

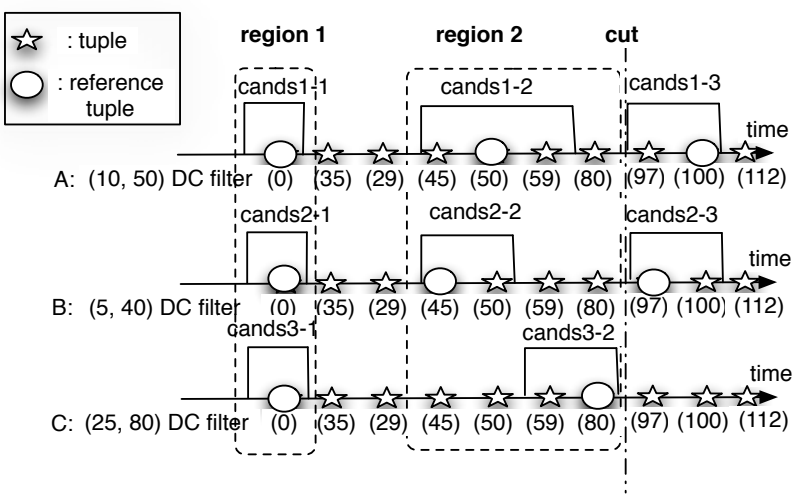
```
REGION-BASED-GREEDY-FILTERING-WITH-CUTS(S)
 1  while ( (currentTuple ← S.getNextTuple()) ! = null)
 2      do for each filter f in the group
 3           do if f.isAdmissible(currentTuple)
 4                then ▷ first stage: admit candidates
 5                     f.candidateSet.add(currentTuple)
 6                     f.state.update(currentTuple)
                       ▷ increment group utility of currentTuple
 7                     globalState.groupUtility.increment(currentTuple)
                  ▷ test if group time constraint is violated
 8                if (globalState.getRegionSpan() ≥ globalState.predictedTime())
 9                  then ▷ force all filters to close their open candidate sets
10                       for each filter f in the group
11                            do f.candidateSet.close()
12                if f.candidateSet.isClosed(currentTuple)
13                  then globalState.addClosedCandidateSet(f.candidateSet)
          ▷ second stage: if current region is ready, decide output based on globalState
14        if (region ← globalState.getCurrentRegion()) ! = null)
15          then ▷ start the timer
16               globalState.startTimer()
                 ▷ apply greedy algorithm to the region to decide the output
17               output ← GREEDY-HITTING-SET(region, globalState.groupUtility)
                 ▷ add output to globalState for multicasting
18               multicaster.send(output)
19               region.currTime ← globalState.stopTimer()
20               globalState.recomputeGroupTime(region.currTime, region.size)
21               globalState.resetTimer()
```

Figure 3.3: Region-based greedy algorithm with timely cuts.

time slot 7 filter C's candidate set still admits the just-arrived tuple 80, yet according to the predicted run-time of the algorithm, admitting another tuple will violate the time constraint, thus a timely cut takes place right before time slot 8. The cut forces cands3-2 to close and consequently region 2 to close. By running the greedy hitting-set algorithm, tuple 59 is chosen for A and C, and tuple 50 for B.

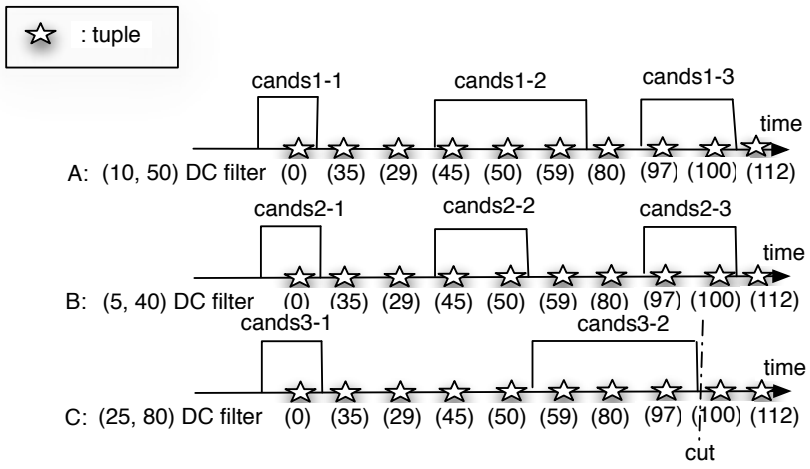
Figure 3.5 shows how a timely cut can be applied with the per-candidate-set based greedy algorithm as well. At time slot 9 filter C admits the new tuple 97, and then checks the time constraint and finds that admitting a new tuple will likely to violate the time constraint. Filter C thus makes a timely cut, closes cands3-2, and chooses tuple 97 as the output, as it has the highest group utility, as the second heuristics applies. Once tuple 97 is chosen by filter C, this information is communicated to other filters via the global state. Thus filter A and B both choose the tuple as their output as well, as the first heuristic of the algorithm applies.

Effect on optimality of the solution It is not hard to see that cuts may adversely affect the optimality in the solution by increasing the output, as a region without cut is the minimum unit of batches to preserve optimality of the solution. How bad can cuts affect the group-aware filtering's solution? As we are interested in promoting group-aware filtering in comparison to self-interested filtering, it should be easy to see that in the worse case, timely cuts lead the sizes of all candidate sets to be 1, which will be exactly the case of self-interested filtering. So, group-aware filtering with cuts should never perform worse than self-interested filtering, in terms of bandwidth savings. The time overhead of enforcing cuts may not be negligible, though, as we will evaluate in the next chapter.



time	1	2	3	4	5	6	7	8	9	10
tuple	0	35	29	45	50	59	80	97	100	112
group utility	3	0	0	2	2	2	1	2	2	0
region status		closed					closed due to cut			closed
chosen outputs		0->{A, B, C}					59-> {A,C} 50->{B}			100-> {A,B}

Figure 3.4: Region-based greedy algorithm with a timely cut for three DC filters.



time	1	2	3	4	5	6	7	8	9	10
tuple	0	35	29	45	50	59	80	97	100	112
group utility	3	0	0	2	2	2	1	3	2	0
closed cand. sets		A, B, C				B	A	C, due to cut		A,B
chosen outputs		0->{A, B, C}				50->{B}	50->{A}	97->{C}		97-> {A,B}

Figure 3.5: Per-candidate-set based greedy algorithm with a timely cut for three DC filters.

3.4 Output strategy

The other factor that affects data timeliness in group-aware filtering is the output scheduling strategy. It enforces different output patterns based on applications' requirements.

First, by computing regions, we get the *earliest possible* time for output tuples of a region without hurting the optimality of the solution. Second, by enforcing group time constraints, we get the *earliest possible time subject to group time constraint* output pattern. Third, filters may opt for a *batched output* pattern, that is, for a fixed-sized (time or tuple) batch of the input stream, select and output tuples.

In the case of a group with all stateful filters, it may be desirable to output tuples at the time any candidate set is closed, if the applications can tolerate disordered output within the predefined time frame. We call this the *per-candidate set* output pattern. The benefit of using this pattern is that the delay of the average tuple is less than that with a region-based earliest possible output strategy. The downside is that it may cause disorder in the output for the candidate sets in a region. Such data disorder can be communicated to the downstream operators via stream "punctuations" [47], control information mixed in the output stream. Quantifying the data disorder and reducing data disorder are interesting future work.

3.5 Related work

We discuss related work in enforcing timeliness in the following areas: first, we discuss requirement analysis and engineering related to data timeliness; second, we discuss mechanisms used for timeliness control especially in real-time systems and other streaming data systems; finally, we relate adaptive control to our mechanism for time prediction.

3.5.1 Requirements engineering

The two key observations that motivate our group-aware filtering approach come from a deep examination of the requirements of monitoring applications. Such requirement analysis is part of requirements engineering, a branch of software engineering concerned with identifying stakeholders and their needs, and documenting them in a form amenable to analysis, communication, and implementation. One challenge of collecting requirements from users is that many goals may be difficult to articulate. Through use cases, devised scenarios and example-based analysis, an understanding of the data-quality requirements of monitoring applications leads to intuition for group-aware filtering. Other techniques for elicitation of requirements include traditional surveys, interviews, brain-storming meetings, focus groups and workshops. Prototyping [51, 72] is a good approach when there is a great deal of uncertainty in the requirements, or when feedback from stakeholders is desirable. To get feedback from real users in the future, our prototype system is ready to be deployed.

To specify requirements, formal logic, semi-formal language, or natural language can be used [51]. We model the time requirement of each filter as a simple threshold, and model that of the group as a conjunction of the time requirements of all the filters in the group. Requirements also need to evolve as the environment in which the systems run can change or users' needs change. Managing evolving requirements is a hot area in requirements engineering. Policy-based specifications have been widely used to manage networks in many different conditions [62]. For quality requirements specified in an absolute, clear-cut way, Letier et al. [40] propose techniques for reasoning about partial goal satisfaction for requirements. Their approach uses goal-refinement models with probabilistic conditioning; the specifications are specified with application-specific measures and alternative goal refinements are evaluated by the refinement equations over random variables involved in the system's functional goals. For group-aware filtering, when the computation resource

and network condition deteriorate to a certain degree, monitoring applications may want to specify how data quality requirements degrade, with policies or probabilistic-based refined requirements.

Integrating requirements from multiple stakeholders, or group-oriented optimizations, may not be trivial, as there may exist conflicts between applications from different domains and each application may have its own priority. Besides negotiating requirements based on modelling each stakeholder's contribution into a single consistent model [57], Beohm proposed a win-win approach [12] where the win conditions for each stakeholder are identified and through negotiation among the stakeholders, the system is managed and measured to ensure all the win conditions are satisfied. For negotiation, the goal is to identify the most important goals of each participant and make sure those goals are met. For group-aware filtering, we assume that satisfying data granularity and data timeliness requirements are the most important goals for each filter and that filters are collaborating for the common goal of saving network bandwidth. We use a group utility to coordinate the filtering for win-win filtering.

3.5.2 Mechanisms for controlling timeliness

Timeliness is a common requirement in most computing systems. In real-time systems that serve mission-critical applications, satisfying timeliness is a core requirement. The requirements usually include tasks' release time and deadlines. Scheduling algorithms for meeting time requirements have been studied extensively for distributed systems with end-to-end time constraints (often called flow shops) [11], pipeline scheduling and distributed tasks with precedence constraints [68]. The general problem of scheduling to minimize completion time on three processors is NP-hard. Thus, most research in scheduling problems is concerned with restricting the problem to make it tractable, or devising heuristic algorithms

to solve the problem approximately. Flow-shop scheduling considers a single sequence of processors, while pipeline scheduling deals with scheduling multiple pipelines to maximize throughput. Pipeline scheduling is thus more complex than flow-shop scheduling as it involves analyzing more complex dependencies among the tasks. Besides deadlines, scheduling algorithms also need to consider the communication overhead between tasks.

For real-time communication, time constraints address the delay of messages from being sent to being received. The message is discarded when the delay exceeds the constraint. Typical applications of real-time communication are packetized audio or video transmission. Such applications require careful encoding and packeting schemes to deal with occasional loss of packets, such that packet loss has minimal effect on the quality of the received audio or video. Other research focuses on coordinating the schedulers at multiple components to ensure that message transmission through the multiple links in a network satisfy deadlines [13]. The increase in the number of choices results in more complicated algorithms with uncertain performance.

Many "soft" real-time systems treat deadlines as a tool to express performance goals. For soft scheduling the performance is measured by the percentage of deadlines it meets. Many soft real-time systems have a global scheduler that schedules cooperative components or nodes in a closed system. For open systems built from pre-existing components of different nature, where control of local schedulers to meet the global timing goals is impossible, Kao et al. [35] consider using on-line techniques such as subtask deadline assignment, server prioritization and real-time emulation to make a cost-performance tradeoff. For overlay systems, where a dynamic number of tasks share one node, it is hard to predict performance of a task accurately. Leveraging multi-core technologies [64] to separate the running of processes on one node will help. Our group-aware filtering uses soft deadlines for timeliness control, yet unlike many scheduling problems, it is concerned with a recurring filtering process with potentially infinite data sources.

Most of the existing stream-processing systems use soft deadlines for exploratory applications, yet the task is semantically different from traditional sense of a task in that the task is recurring on potentially infinite data sources. Thus, deadlines are discussed in terms of a task within a finite data window. Sometimes, rather than defining deadlines, applications specify QoS functions regarding how well the performance meets the desired quality of service [7, 8].

Dynamic load shedding [65] is a common technique to deal with a bursty data load. Pipeline scheduling among multiple operators [15] is also a hot research area. The timeliness of our group-aware filtering is also affected by the size of the batch the filters process at a time, but unlike most of stream-processing systems, the batch size is not predetermined. It depends on many on-line factors based on applications' requirements and characteristics of the data being processed, as the timeliness is dictated by when a candidate set closes, or when a region closes.

3.5.3 Adaptive control

In enforcing timeliness in region-based group-aware algorithms, we use online monitoring-based prediction for the algorithm to choose the time to stop accumulating tuples and to produce outputs. The purpose is to adapt to the changing rate of the incoming data and adapt to the changing processing rate of the batched data in a region. Such adaptive control is often used for controlling of a dynamic system.

Adaptive control addresses the ability of a system to modify its behavior based on the performance of a closed-loop system. There have been prolific results in theory on this subject. Based on stability theory developed in the 1950s, and optimal control in the 1960s, as part of control theory, and became a mature subject in the mid 1980s. Yet, research on the practical use of adaptive controls emphasizes the robustness of control algorithms and

the translation of design rules to understandable settings for the designer and operator [28]. It has been observed that it is often the strategy of a controller which is important and leads to economic savings; the adaptation only being an implementation aspect that tunes the parameters of a specific process. Two main types of adaptive controllers are model reference and self-tuning adaptive control. A "model-reference" controller uses the desired reference model explicitly to compute the desired model output in response to the reference input. The output is then compared to the actual output leading to an error. The controller then computes a compensator to adopt in the model. A model-reference controller is most useful when the desired behavior of the system is known. A "self-tuning" controller uses run-time measurements to compensate a model. For our timeliness enforcement, we use the self-tuning time model with online measurements to estimate the algorithm's running time dynamically.

Chapter 4

Evaluation

This chapter focuses on performance evaluation of group-aware stream filtering with delta-compression filters, in comparison to self-interested filtering, in terms of network bandwidth consumption and its effect on data timeliness.

4.1 Prototype system

We use a systems approach to evaluate the group-aware stream filtering. Below we describe the software architecture of our prototype system, and how it is integrated with the application-level multicast service of a general data-dissemination system, Solar. We then describe the setup of the experiments.

4.1.1 Software architecture

The software architecture of the framework shown in Figure 4.1 consists of the following modules: 1) the quality specification manager, which facilitates applications to specify their data quality needs with a library of common predicates, distance functions and member functions, 2) the group-aware stream filtering manager, which instantiates and coordinates

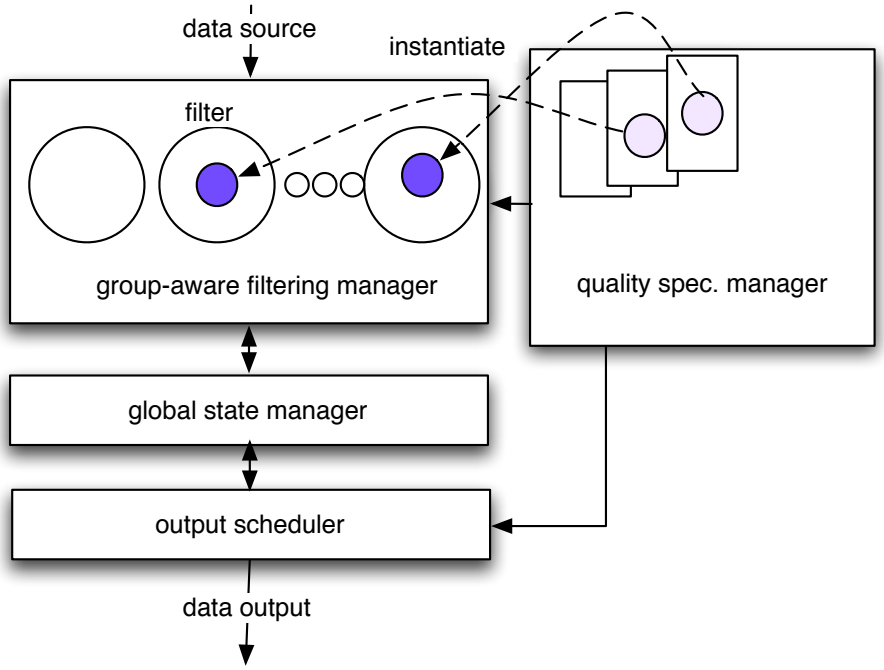


Figure 4.1: Framework for group-aware stream filtering.

a pool of filters for group-aware filtering based on quality specifications, 3) a global state manager that maintains the state information shared by the filters, and 4) an output scheduler that multiplexes the decided outputs of the filters and schedules the outputs with an application-level data multicasting protocol. It monitors the data-filtering progress via the global state manager, and schedules outputs according to the group's output strategy. For instance, if the group's time constraint is about to be violated, the output scheduler signals (via the global state manager) the filters to close their current candidate sets and decide outputs.

We implement and integrate the group-aware filtering with *Solar* [21], a general-purpose data dissemination system developed at Dartmouth College. The core of Solar is a p2p overlay infrastructure in which each overlay node supports a suite of data-dissemination services, such as naming, data fusion, and multicasting. We package the group-aware fil-

tering as a new service at the application layer, working together with Solar's basic services on each overlay node.

Solar uses a content-based publish/subscribe model for flexible and scalable data dissemination. Publishers of context sources in Solar are called "sources" and applications can "subscribe" to sources in Solar to get the desired context information. Solar also allows an application to specify data operators, such as filters for pre-processing the source data. Solar disseminates events with an application-level multicast facility built on top of its peer-to-peer distributed hash table-based routing substrate (Scribe [16]).

4.1.2 Experiment setup

For our testing we replay real-world data traces as Solar sources and let a group of applications subscribe to the sources. Each subscribing application specifies a filter for its processing needs. The group-aware filtering service then deploys, according to a filter's type and quality requirements, a group-aware filter object on the source node. The union of the output of all source-sharing filters is published via Solar's overlay multicasting service to the remote applications.

To compute end-to-end latency based on time stamps, we deploy the subscribing applications on the same node as the data source to eliminate time skew in a network. Here we assume the real end-to-end latency is the time difference we will measure between a tuple published from the source and the time it arrives in an application, plus a constant number that captures overlay multicasting cost.

To get a basic idea of the cost of using overlay multicasting, we deployed Solar in Emulab [1] with a small (7-node) overlay network with 1 Mbps capacity on each link and measured that Solar's overlay multicasting delay was about 130 ms [41, 42].

[1] http://www.emulab.net is a cluster for distributed-systems research

We deployed Solar on a node that hosts the data sources for testing. The computer is an Apple Powerbook with 1.67 GHz PowerPC G4 and 1 GB memory. Our code is written in Java and run with Java 1.5.0 on Mac OS 10.4.11. We deployed group-aware filters on the same node and multicast the output to applications on remote nodes.

4.2 Data sources

We chose data from real deployments of sensing devices for which the data stream has a sub-second data rate, so filtering is necessary and saving bandwidth for dissemination of the data is important.

The Networked Aquatic Microbial System (NAMOS) of the CENS project at UCLA[2] deployed embedded and networked sensors in Lake Fulmor for a marine scientific study during August 2006. The water was monitored by an array of thermistors and fluorometers, among others, installed on buoys of the lake. The data traces have data rates of 100 measurements per second and contain more than ten thousand measurements. These measurement traces make ideal data sources for our testing. Each NAMOS buoy trace tuple contains six temperature readings (we call them *tmpr* readings), one reading from a fluorometer (we call it the *fluoro* reading), a timestamp, and some other weather-related readings. We create a source in Solar that replays the NAMOS buoy trace at about 10 ms per tuple, observing the original time intervals of the trace data.

We use NAMOS buoy traces as the primary data source for our evaluation. In the later section of the chapter, we use several other real-world and synthetic data sources, such as seismic data collected from a wireless sensor network deployed around a volcano [69], data collected from fire experimentation from fire study program at WPI [54], and synthetic data traces for a carefully modeled chemical spill in an emergency response study, to evaluate

[2]http://cens.ucla.edu

the effect of data sources on the performance.

4.3 Filters for testing

The goal of the NAMOS buoy deployment is to help marine biologists collect multi-scale high-resolution information, such as the spatial and temporal distribution of the chlorophyll level in the lake, for scientific analysis. Using delta-compression filters is a valid way to enforce multi-scale granularity of the collected buoy data for these applications. In this chapter we focus on the performance evaluation of group-aware Delta-Compression (DC) filters. In the next chapter, we show how our framework accommodates other types of filters, such as stratified sampling operators.

Each DC filter has three parameters: the data attribute(s) that the filter is interested in, a delta value for compression, and a corresponding quality slack it can tolerate. Table 5.2 shows the groups of filters we used for our testing. To set parameters for the *DC_Fluoro* and *DC_Tmpr* filter groups, we computed the average changes, *srcStatistics*, of two consecutive tuples in the source time series and then randomly picked delta values between the range of *srcStatistics* and 3**srcStatistics*, which ensured a reasonable data compression that had a non-trivial output data volume. Then we set slack values to be about 50% of the corresponding delta values. This prevented a tuple from being included in more than one candidate set for a filter, and also ensured large candidate sets for us to see the benefit of group-aware filtering. For the DC_Hybrid filter group we randomly picked delta values from the range of *srcStatistics* and 20**srcStatistics* and randomly picked slack values that were less than 50% of corresponding delta values. Below, we also evaluate slack's effect on the performance of the delta-compression filtering.

GROUP NAME	FILTER
DC_Fluoro	DC(fluoro, 0.0301, 0.0150)
	DC(fluoro, 0.0702, 0.0301)
	DC(fluoro, 0.0500, 0.0250)
DC_Hybrid	DC(fluoro, 0.0702, 0.0100)
	DC(tmpr2, 0.0460, 0.0153)
	DC(tmpr4, 0.0310, 0.0103)
DC_Tmpr	DC(tmpr4, 0.0310, 0.0155)
	DC(tmpr4, 0.0620, 0.0310)
	DC(tmpr4, 0.0480, 0.0240)

Table 4.1: Specifications for groups of filters.

4.4 Metrics and basic results

The metric we use to measure the benefit of our group-aware filtering approach is the *O/I ratio*, that is, the output vs. input ratio defined as the total number of output tuples over the number of input tuples. A lower O/I ratio means low bandwidth consumption. It measures the bandwidth-saving benefit of group-aware filtering. We expect group-filtering should have an O/I ratio no more than that of self-interested filtering. We measured the filtering cost with *CPU time per tuple*, representing the CPU overhead of group-aware filtering. We measured data timeliness with source-to-application *latency per tuple*, which shows the delay induced by group-aware filtering to each output tuple. Again, this thesis does not focus on the network aspects of group-aware filtering and we do not measure network behavior while performing group-aware filtering.

Table 4.2 shows the notations we use for filters in the results. Figure 4.2 shows the O/I ratios for three groups of filters. All group-aware filtering algorithms consumed less than 80% of the bandwidth consumed by self-interested filters. PS filters had a performance comparable to RG filters, which in theory should have a better performance guarantee. The addition of timely cuts had little impact on O/I ratio in this experiment, as we set the group time constraint large enough so that few regions were cut.

ABBREVIATION	MEANING
SI	Self-Interested filter
RG	Region-based Greedy filter
PS	Per-candidate-Set greedy filter
+C	with timely Cuts
+C(x)	with timely Cuts, x is the name of a time spec.
(B)	with Batched output strategy
(B)-x	with Batched output strategy, x is input tuple window
(Pcs)	with Per-candidate-set output strategy

Table 4.2: Filter type notations

Figures 4.3, 4.4, and 4.5 show the CPU cost per tuple for the *DC_Fluoro* group, the *DC_Hybrid* group, and the *DC_Tmpr* group respectively. (The results are in a box-plot, which plots a summary of the minimum, 25% quartile, median, 75% quartile, and maximum of the ten results. The circles represent outliers. Any data observation which lies more than $1.5 \cdot \text{IQR}$ lower than the first quartile or $1.5 \cdot \text{IQR}$ higher than the third quartile is considered an outlier, where IQR is the inter-quartile range by subtracting the 25% quartile from the 75% quartile.) Group-aware filters were more than 10 times more expensive than self-interested filters. However, it took only 1 ms for processing each tuple for group-aware filters, which is fast enough for an input stream with a data rate of 100 tuples per second. Figure 4.6 shows the latency per tuple for the *DC_Fluoro* group. Since the group-aware filtering gathers tuples in a region before releasing output, it is understandable that the latency incurred for group-aware filters (about 70ms per tuple) was much greater than that for self-interested DC filters (about 12ms). The average region size of the filters was about 6 tuples; since tuples arrived at 10ms intervals, it is clear that the 58ms difference of latency was mainly due to waiting for the tuples to arrive for processing in segments. Similar conclusions can be drawn to the latency for the *DC_Hybrid* group (Figure 4.7) and the *DC_Tmpr* group (Figure 4.8).

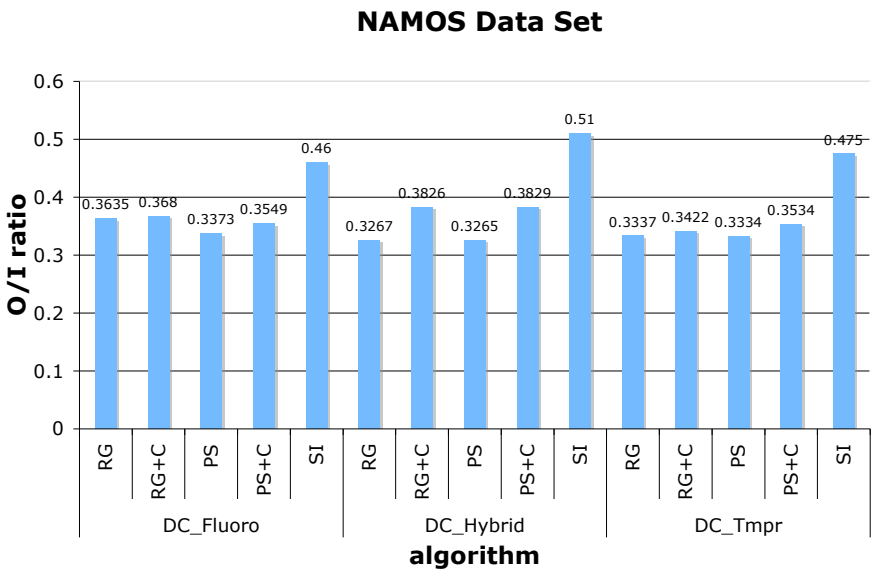


Figure 4.2: O/I ratios for three groups of group-aware filters.

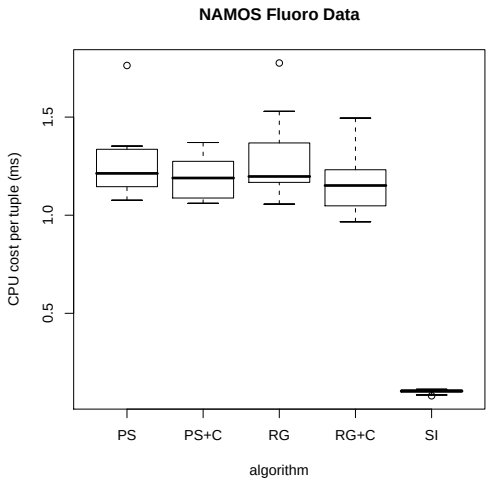


Figure 4.3: CPU cost for DC_Fluoro.

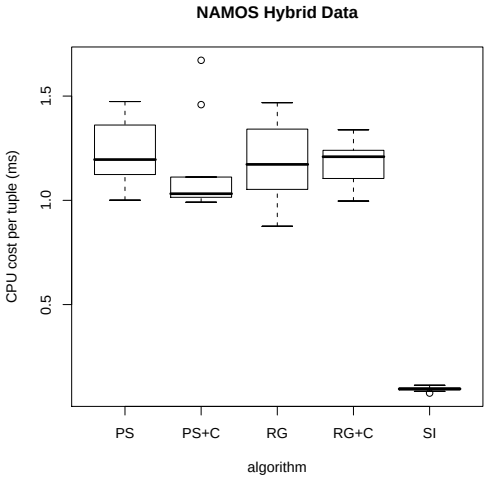


Figure 4.4: CPU cost for DC_Hybrid.

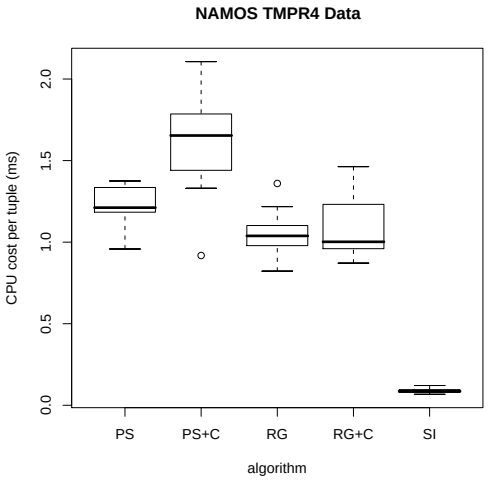


Figure 4.5: CPU cost for DC_Tmpr.

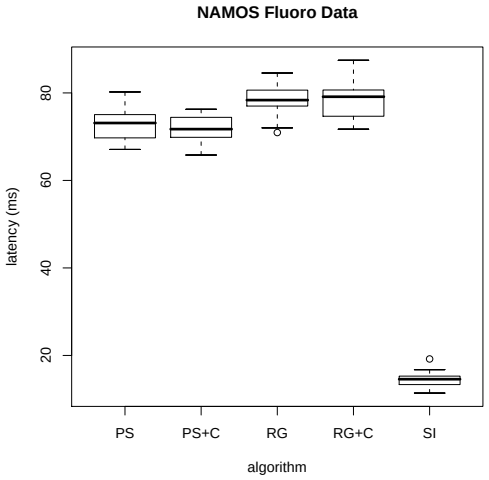


Figure 4.6: Latency for DC_Fluoro.

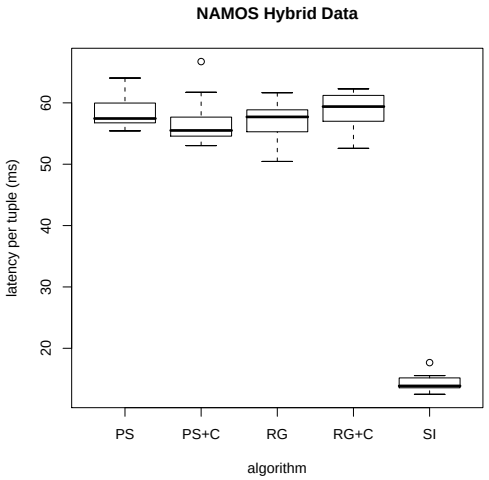


Figure 4.7: Latency for DC_Hybrid.

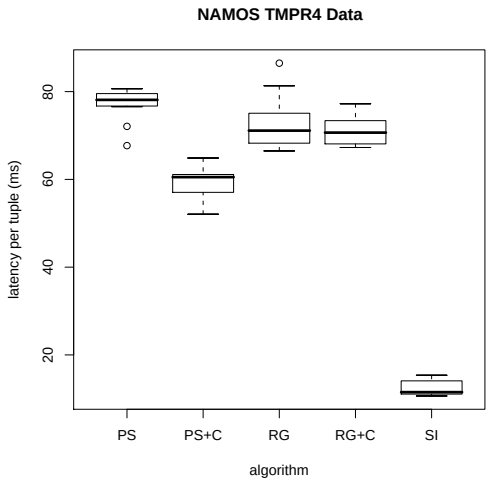


Figure 4.8: Latency for DC_Tmpr.

4.5 Effectiveness of cuts

Next, we compare the performance of algorithms that enforce timely cuts. By linearly decreasing the maximum time for closing a region from 125 ms in $RG+C(01)$ filters, to a time 16-fold less in $RG+C(05)$ (8 ms), the resulting average latency per tuple consistently dropped from above 70 ms per tuple to about 20 ms per tuple (Figure 4.9), thus proving that timely cuts were effective. Figure 4.10 shows that the CPU cost to enforce cuts, less than 0.5 ms per tuple, is acceptable for a fast stream with a 10 ms tuple interval. The percentage of regions cut consistently increased by decreasing the maximum time allowed for a region (Figure 4.11). Cuts affected the O/I ratio slightly (Figure 4.12), which is understandable because cutting a region would affect the optimality of the solutions found and it is a necessary trade-off for a latency-sensitive filter.

4.6 Effect of output strategies

Finally, we evaluate the output strategies with the *DC_Fluoro* filter group (Figure 4.13 and Figure 4.14). The latency was affected mostly by the size of the average region a group-

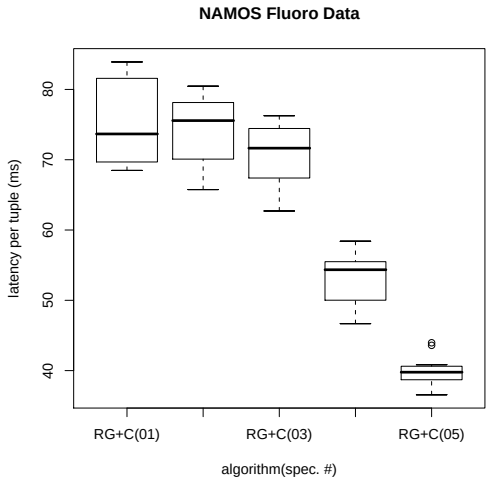


Figure 4.9: Cuts affect latency for DC_Fluoro.

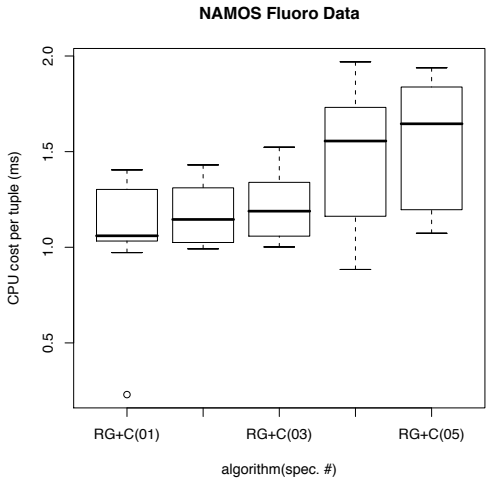


Figure 4.10: CPU cost of cuts for DC_Fluoro.

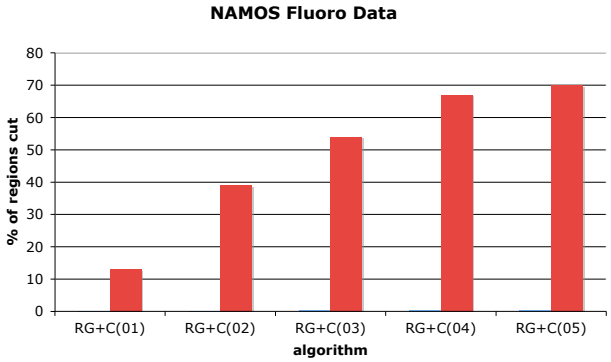


Figure 4.11: Percent of regions cut for DC_Fluoro.

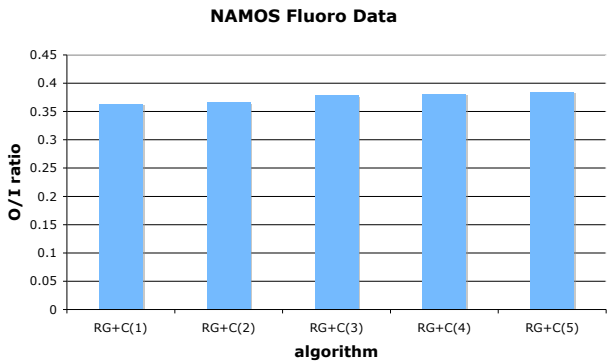


Figure 4.12: Cuts affect O/I ratios in DC_Fluoro.

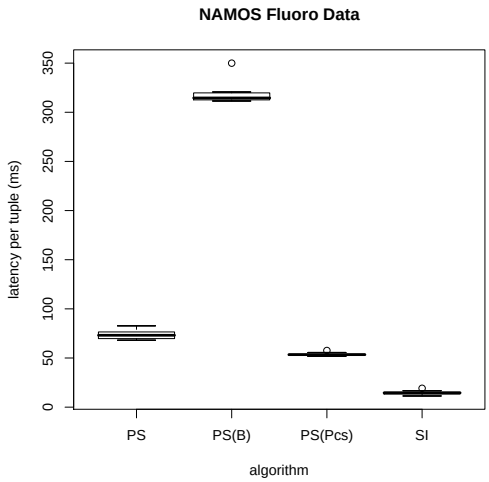


Figure 4.13: Output strategy affects data timeliness.

aware filter used before producing outputs. In the batched output pattern, when the batch size was much bigger than the size of a natural region, the latency increased dramatically due to backlogging of the tuples in the filters until enough tuples were processed. The per-candidate-set output strategy helped to decrease the latency from above 70 ms per tuple to a little above 50 ms per tuple. In terms of CPU cost the batched output pattern did not require sophisticated checking on whether a natural region is closed, which cut 1 ms from the original 1.3 ms CPU time.

4.7 Factors that affect the performance

Factors such as slack, delta, and group size intrinsically affect the performance of group-aware filtering for delta-compression filters. We first evaluate those factors with the filter-group *DC_Tmpr* with the NAMOS data trace. Then we evaluate how different characteristics of a data source affect the performance.

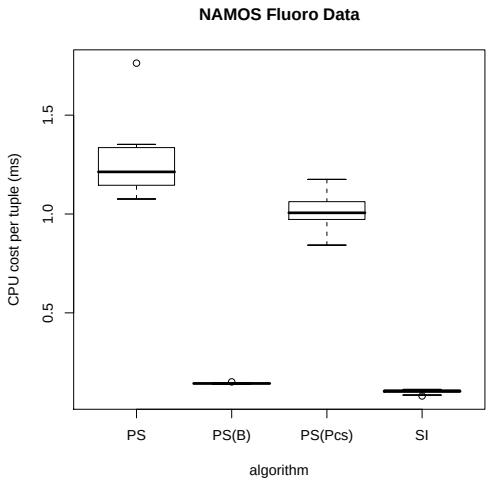


Figure 4.14: CPU cost of output strategies.

4.7.1 Slack of a delta-compression filter

Intuitively, when a delta-compression filter has a larger slack, there are potentially more tuples in a candidate set and thus it is more likely to find candidate sets that overlap with other filters, and reduce the size of the combined output set. We retested filters in *DC_Tmpr* by varying their slacks from 3% to 50% of the corresponding delta values. The results in Figure 4.15 confirm our intuition. When the slacks were 20% of the corresponding delta values, the output ratio was more than 90%; when the slacks increased from 20% of delta to 50% of delta, the output ratio decreased from 90% to below 75%.

4.7.2 Delta of a delta-compression filter

The delta value of a delta-compression filter affects its filtering stride and thus the Number of Candidate Sets (NCS) it has for a fixed data sequence. In general, the bigger the delta value, the fewer candidate sets can be found. Yet, a change in the delta value means changing the potential overlap between candidate sets. It is equally likely for a decreased NCS of a filter to increase or to decrease the total number of outputs of the group, depending on

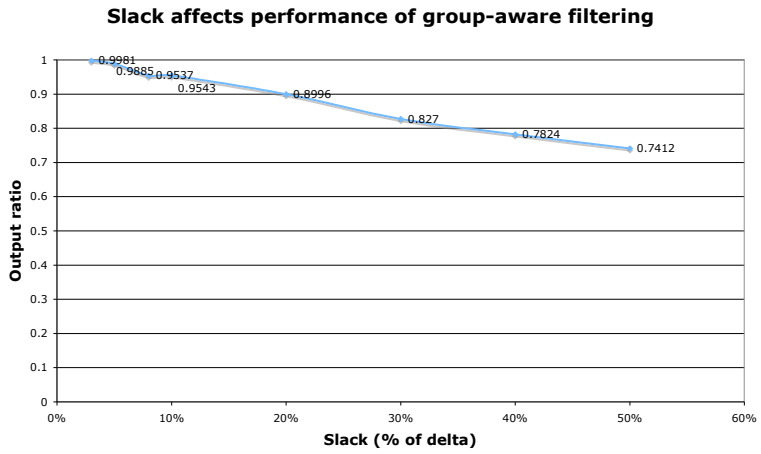


Figure 4.15: Slack's effect on the performance of DC type filters.

how the new candidate sets relate to those of the other filters in the group.

We explored this situation with the group *DC_Tmpr*. We fixed the slack values at 0.0155 (or $50\%\cdot$ *srcStatistics*) for all filters (note that *srcStatistics* is the average change of two consecutive tuples in a data source). We fixed the delta values for two filters at 0.0930 ($2\cdot$ *srcStatistics*) and 0.01340 ($3\cdot$ *srcStatistics*) and randomly generated a number between 0.0310 ($1\cdot$ *srcStatistics*) to 0.0930 ($2\cdot$ *srcStatistics*) as the delta value for the third filter.

Figure 4.16 shows the result of the test. We have a few observations. First, the curve is mostly level with a few outliers. For example, the difference of output ratios was less than 5% for the delta values between 0.050 to 0.075. This difference was probably caused by the big slack value of 0.0155; a large portion of the output tuples of the filter whose delta value was changing were covered by the candidate sets of the other two filters. The sudden increase of 20% in the output ratio when the delta changed from 0.045 to 0.049 can be explained by the fact that many of the output points of the changed filter moved

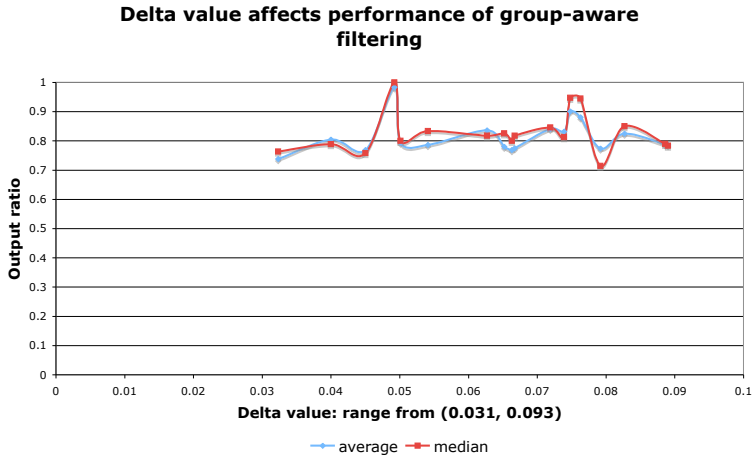


Figure 4.16: Delta's effect on the performance of DC type filters.

out of coverage of the candidate sets of the other two filters. This caused the total number of outputs by the group (and the output ratio) to increase dramatically. Another effect of increasing delta value is that the total number of output tuples by the corresponding self-interested filter may decrease, and if there is no change in the total output by the group-aware filters, the output ratio may go up.

The sudden drop of the output ratio at 0.050 can be explained by the fact that the candidate sets that did not overlap with those of other filters start to overlap and the overlapped data were chosen as output. Also, notice that the average output ratio before the spike at 0.049 is about 0.76 and the average output ratio between 0.05 and 0.075 is about 0.82. The increased output ratio is probably due to the fact that increased delta value caused slightly decreased NCS or number of reference points; the number of overlapped tuples may change very little, yet the total number of outputs chosen by self-interested filters is reduced more significantly in comparison. That makes the output ratio go up slightly.

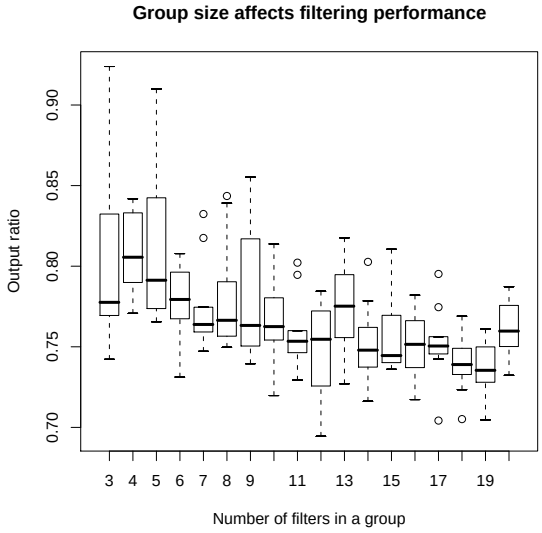


Figure 4.17: Group size's effect on the performance of DC filters.

4.7.3 Group size

The number of filters in a group may also affect the performance of group-aware filtering. We tested the group *DC_Tmpr*. For each group size we generated 10 groups of *DC1* type filters on thermo4 and we fixed the slack value to be 0.015. We randomly chose delta values from the range of 0.031 ($1\cdot$ srcStatistics) to 0.186 ($6\cdot$ srcStatistics). We varied the group size from 3 to 20. Figure 4.17 shows the results in box-plot form.

Overall there is a downward trend in the median of the output ratios for the results: that is, adding more applications to the group seems to decrease the output ratio, because the increase of the total output tuples due to adding new filters was smaller than the increase in overlap among candidate sets. The large slack value of 0.015 made the candidate sets of newly added filters easy to overlap with those of other filters in the group.

Group size also affected the CPU cost of the filtering. Figure 4.18 shows a roughly linear increase of CPU cost per batch of 100 tuples when we increased the number of applications in the group from 3 to 20, both for group-aware filtering and for self-interested

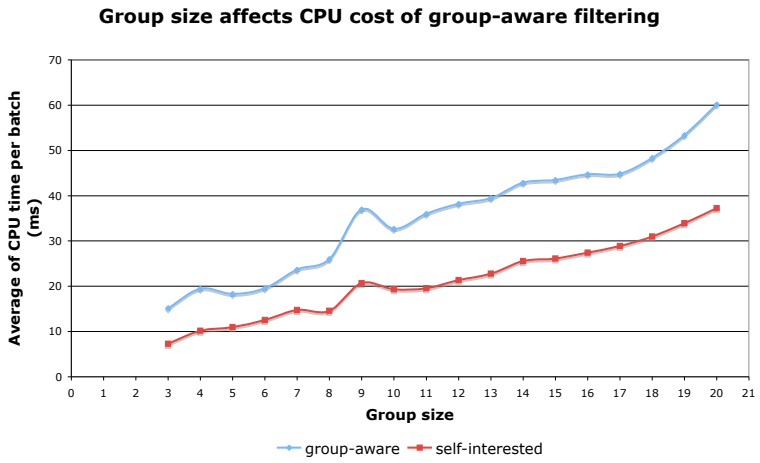


Figure 4.18: Group size's effect on the cost of DC filters.

filtering. The group-aware filtering required about double the CPU cost as self-interested filtering, due to the complexity of group coordination. The CPU cost for any of the groups was less than 61 ms, however, which was low compared with the latency of multicasting.

4.7.4 Source data

The characteristics of a data source directly affects its filters' performances. All our previous evaluation is based on NAMOS buoy data traces. Here, we evaluate the basic performance of group-aware delta-compression filters with three real-world data traces from other domains. The first source is a cow's movement data, specifically its orientation change. The data is collected by a bio-monitoring research group in the EECE department at MIT [59]. The second source is readings of seismic sensors deployed near a volcano in Peru [69]. The third source is chemical readings, specifically HRR(Q) readings, from fire experiments conducted by a group of researchers in fire prevention program at WPI [54].

GROUP NAME	FILTER
DC_cow	DC(E-orient, 8, 4)
	DC(E-orient, 12 ,6)
	DC(E-orient, 16, 8)
DC_volcano	DC(seis, 0.0005, 0.0002)
	DC(seis, 0.0011, 0.0006)
	DC(seis, 0.0007, 0.0003)
DC_fireExp	DC(HRR, 0.10, 0.05)
	DC(HRR, 0.15, 0.07)
	DC(HRR, 0.21, 0.10)

Figure 4.19: Filter specifications for multiple data sources

Table 4.19 shows the delta-compression filters we specify for each data source.

We set the delta and slack values of filters in the same manner as previous experiments: we compute the average changes, *srcStatistics*, of two consecutive tuples in the source time series and then randomly picked delta values between the range of *srcStatistics* and 3**srcStatistics*, which ensured a reasonable data compression that had a non-trivial output data volume. Then we set slack values to be about 50% of the corresponding delta values.

Figure 4.20 shows the O/I ratios of running group-aware filtering on each of the three real-life data traces. On each data source, there was an additional bandwidth savings with group-aware filtering. Group-aware filtering reduced bandwidth consumption to 83%, 74%, and 60% of that with self-interested filtering for cow's movement, seismic readings, and fire HRR(Q) trace respectively.

The difference in the bandwidth savings is mainly determined by the value's update patterns in the time series. Figure 4.21, Figure 4.22, and Figure 4.23 show the curve for the cow's orientation changes, seismic readings for the volcano, and the HRR(Q) reading changes in the fire experiment respectively. They are distinctive in the shape. The cow's orientation presents clustered brief changes over time. The seismic update pattern and

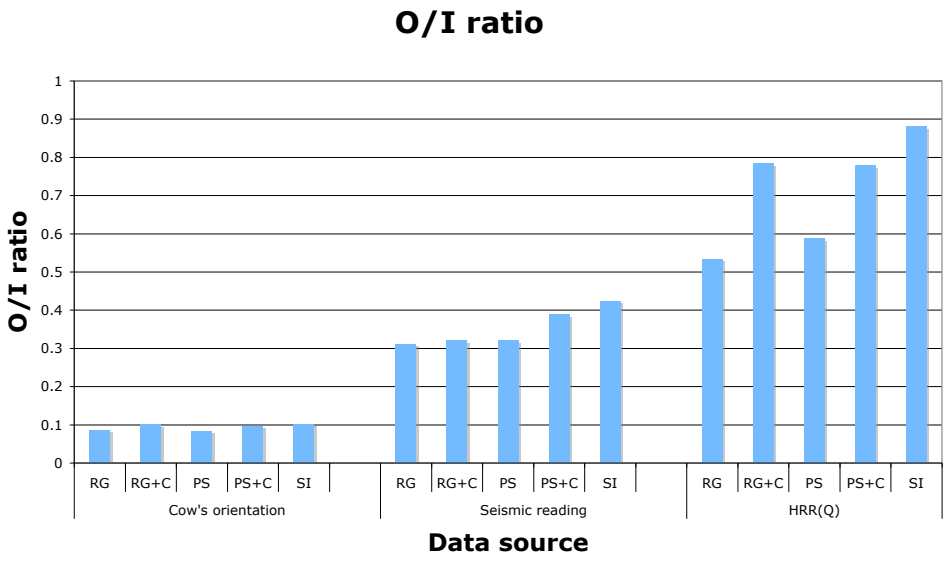


Figure 4.20: O/I ratios of filtering with different data sources

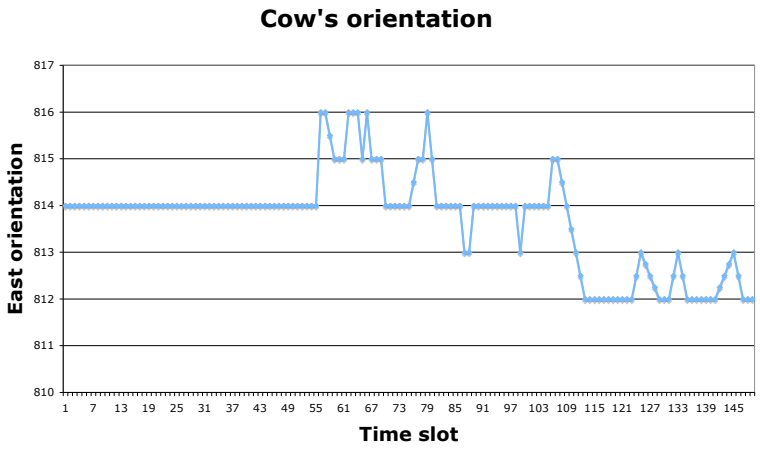


Figure 4.21: Cow's orientation changes

HRR(Q) change pattern are relatively smoother curves.

All timely cuts adversely affect O/I ratio (Figure 4.20). The cuts' effect is negligible for the data source of the cow's orientations. It significantly hurt the bandwidth-saving performance in the case of the HRR(Q) trace. This can happen when most of the original overlapping candidate sets are cut and no longer connected, and thus removing most of the benefit of group-aware filtering. The seismic readings is a case in between the two extremes. The cuts moderately affected the bandwidth saving.

Figure 4.24 shows the CPU cost of running each of the group-aware filtering algorithms upon the data sources. All group-aware filtering algorithms raised CPU cost, compared with self-interested filtering. But the additional CPU cost was less than 50% added CPU cost for each data source. Again, since each data source has distinct update patterns, the overlapping patterns of the candidate sets were different. They not only affected the absolute running time of each algorithm, but also affected the relative cost of timely cuts.

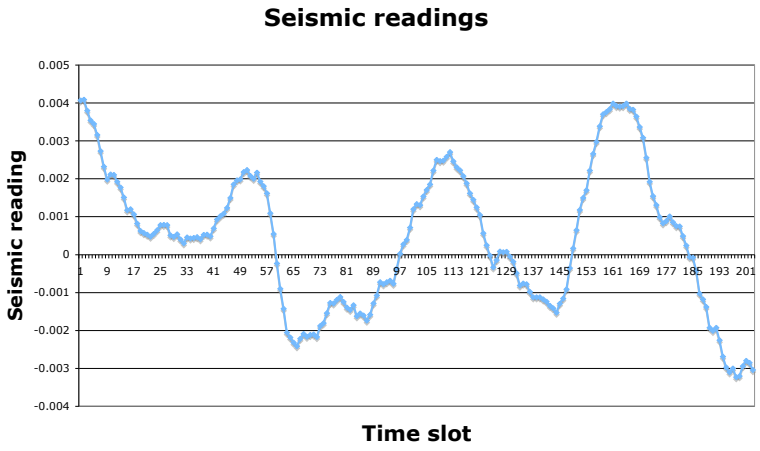


Figure 4.22: Seismic updates for a volcano

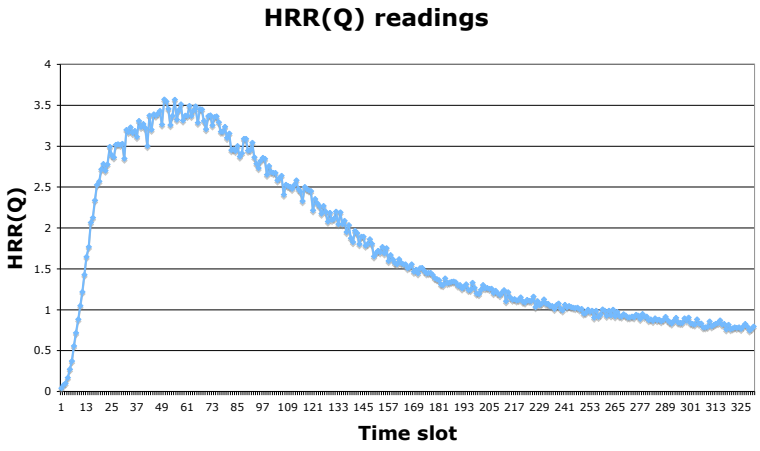


Figure 4.23: HRR(Q) updates in a fire experiment

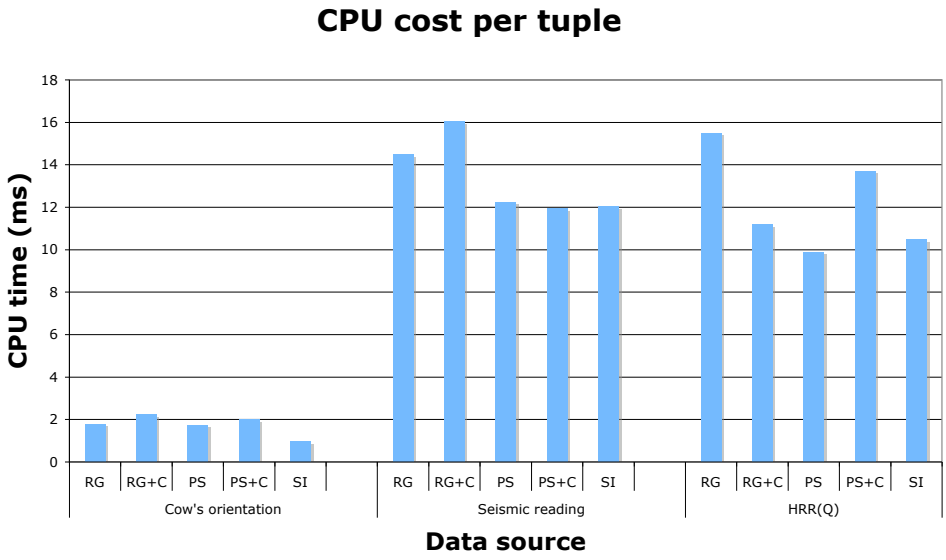


Figure 4.24: CPU cost of filtering with different data sources

For example, for the data capturing cow's orientation changes, RG+C had more overhead than RG, and PS+C had more overhead than PS. This difference is because checking time constraints after each tuple had extra cost and that cost could not be compensated for by the savings in CPU due to shrunk regions or candidate sets. For the seismic readings, however, PS+C cost less than PS, because the CPU savings due to shrunk candidate sets was more than the cost of having time constraints checked. The same was true for the case of RG+C vs. RG in the HRR(Q) of a fire experiment.

4.8 Discussion

We get a glimpse of the performance of group-aware delta-compression filtering from the above experiments with multiple real-life data traces. In most cases, we found that the group-aware filtering could reduce the bandwidth demand below 80% (or lower) of the original bandwidth demand of the self-interested filtering. The application-level multicast latency is dominated by the software delay of invoking the multicast protocol, rather than the data transmission or propagation delay. The CPU overhead was low and negligible compared with the latency of overlay multicasting in a wireless network. We conclude that group-aware filtering is likely to be a valuable approach for saving bandwidth.

Our experiments confirm some important properties of our group-aware filtering algorithms. First, group-aware filtering works no worse than self-interested filtering in terms of bandwidth consumption. Second, for moderate group size, CPU overhead is small and thus can be negligible in the overall latency caused to the streaming data. Yet, with a large group size, the overhead can cause congestion at the input buffer of the filter. The system needs to resort to other mechanisms to resolve it. For example, Solar installs flow-control filters in the buffer to alleviate congestion [21]. The system may also employ more aggressive sampling to shed data load, or gracefully degrade the quality requirements of the filters in order to keep up with the flow. Third, cuts are effective in reducing latency due to batch processing in group-aware filtering, but cuts almost always adversely affect bandwidth savings. Again, cuts perform no worse than self-interested filtering in terms of bandwidth savings. And it may be equally likely to increase or decrease CPU cost of running the group-aware filtering algorithms.

Our evaluations also show that it is quite hard to predict exactly how group-aware filters perform, mainly due to the fact that the bandwidth-saving benefit is subject to the interplay of many factors, such as data update patterns in the source data, the delta and slack values

and the the overlapping patterns of the candidate sets of a group of filters. It is thus important to resort to on-line monitoring of source data and current performance to get a hint as to how group-aware filters can benefit. We measured the CPU costs in our evaluations with the assumption that the co-existing work load at the nodes where group-aware filters run is small and constant. In real deployment, the nodes that host the filters may host many unrelated processes with dynamic run-time behavior. Thus, predicting running time of the algorithms is difficult. With multi-core technologies advancing, it may be possible in the near future to devote a whole processor to the filtering, separating the filtering from other running processes.

Our evaluation and analysis point to some problems and interesting directions for improving the system. For example, when a group has a filter that requires most of the data from the source, group-aware filtering will not save much bandwidth at the output link of the source node, or worse, the CPU overhead of group-aware filtering might cause congestion at the input buffer and the filtering will degrade data timeliness unnecessarily. It is desirable to isolate those "bad" filters from the rest, or not to apply group-aware filtering when they are present. It is thus important to monitor the selectivity of each filter. Another way to alleviate the congestion-causing effect of group-aware filtering is to reduce the group size. Large groups increase CPU overhead and, in some cases, may violate the latency constraints of some of the filters or cause input congestion. We thus need to develop strategies for (re)grouping the filters. Grouping applications according to their locations (within the network topology) may reduce multicast overhead due to simplified groups. All of these optimizations represent future work.

Chapter 5

Extensible Framework

In this chapter, we illustrate a taxonomy of group-aware filters, based on diverse filtering needs of monitoring applications. We describe a general framework that supports flexible extensions to the group-aware filtering approach to meet the applications' needs. Then, we evaluate the basic performance of a set of diverse types of group-aware filters based on the NAMOS buoy data trace. Finally, we discuss related work.

5.1 Diverse filtering needs

We treat any data-selection operator that picks out a subset of the data contained in a source stream as a *filter*. Here we discuss how group-aware filtering supports a variety of filters beyond simple delta-compression filters.

Slack-based filters. Group-aware filtering is motivated by applications that tolerate slack in their quality requirements. Those slack-based filters provides a way to compute multiple candidate outputs around each reference output. The slack-based filters used in the previous chapters are simple in that the quality slack is directly computed based on the values of one

of the attributes of the time series. For instance, the DC_Fluoro filters are a group of filters where the filtering is based on the values of a single attribute $Fluoro$ of the NAMOS buoy time series.

Slack-based filters may also compute candidate sets by keeping track of some internal states based on domain-specific functions. For instance, if an application is interested in the changing rates or the "trends" of temperature values, the filter may want to compute the ratio of the temperature change over a time span for each tuple. If the ratio is greater than some threshold, it admits the tuple in question to the candidate set.

Here is another example of slack-based filters, whose candidate-set computation involves multiple attributes of the data. If a data stream consists of readings from multiple sensors of similar sensing capacities deployed in close vicinity, a filter may compute the "averaged" readings over multiple attributes of the source data to see if it is above some threshold value. If it is, the tuple in question is admitted to the candidate set.

The distance functions used for computing the thresholds for admitting candidates can be complex. For example, if a tuple contains two-dimension coordinates of a location, the natural distance function will be Euclidean distance. Also, for classification-based candidate admission, domain-specific membership functions, such as fuzzy rules for "safe" zones, may be used.

Sampling filters. For many exploratory data-analysis applications, *sampling* filters are of special interest. Sampling filters derive interesting properties by choosing a small set of data from a population. The notion of candidate sets is inherent in many commonly-used sampling methods, such as reservoir sampling, subset-sum sampling and stratified sampling [66, 33]. For example, reservoir sampling chooses a fixed number of samples from a given population. Each tuple in the result can be replaced randomly by another tuple in the population. In this case, the candidate set of each output tuple is the whole data

sequence in a predefined window. Reservoir sampling can be useful to bound the output bandwidth demands for some applications. For detection-oriented analysis, predicates that recognize interesting patterns can first be applied to the time series to distinguish important data sequences from less important ones, and then a higher sample rate can be applied to the more important data segments. This sampling theme belongs to *stratified sampling*, as it first decides strata of data with different characteristics and then samples each stratum with a different sample rate.

Rules for equivalence in quality. Some monitoring applications may employ sophisticated rules to define quality equivalence among tuples. For instance, given a data source that contains sensed information from many distributed sensing devices, the application may treat as equivalent in quality any tuples that come from devices that are "close" in physical and temporal distance and in sensing capacity. Reasoning on the closeness and similarity of the properties of the data is domain-specific.

5.2 Taxonomy of group-aware filters

We develop a taxonomy of filters supported by the group-aware filtering framework, along the following three dimensions: candidate computation, output selection, and dependency of candidate sets (see Figure 5.1).

Candidate computation determines how candidates are computed. It consists of three elements: first, a list of attributes of the source data that a filter considers; second, a state-update function, which applies to the attributes; third, a threshold function, such as a distance function or a membership function, applied to the internal state of the filter for deciding whether a tuple is a candidate.

The second dimension is *output selection*. It concerns the choice of outputs from a

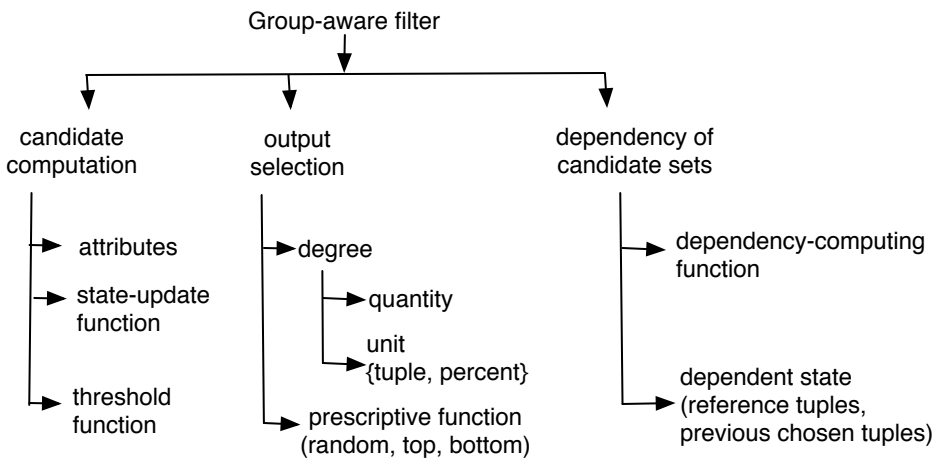


Figure 5.1: Taxonomy of group-aware filters.

candidate set. It includes degree of candidacy, i.e., quantity of outputs either in tuple counts or in percentage of tuples included in a candidate set. For example, specifications for output selection might include "pick 2 tuples out of each candidate set", or "pick 40% of the tuples from each candidate set". Output selection also considers how outputs are decided, which is specified by domain-specific prescriptions, such as prescriptive functions "top","bottom", or "random" (as the default value). For example, "pick the top 3 tuples with regard to values in the attribute X from a candidate set", "pick the bottom 5% tuples with regard to values in the attribute Y from a candidate set", or "pick 5% random tuples from a candidate set."

The third dimension, *dependency of candidate sets*, indicates how candidate sets are dependent on each other, specifically whether candidate-set computation is based on reference tuples, or previous chosen outputs. If it is based on previous outputs, what function should we use to compute the dependency? For example, for a stateful (5,20) delta-compression filter, we can specify its *dependent state* as "outputs from the previous candidate set" and its *dependency-computing* function to check whether a tuple is within $(20 - 5 =)15$ and $(20 + 5 =)25$ unit distance away from the last output.

5.3 Framework for extensions

The two-stage process of group-aware filtering is general enough to cover the diverse needs of the filters. Our framework provides many mechanisms for applications to flexibly extend the two-stage process to enforce their filtering requirements.

First, in the group-aware filtering service package we include a library of distance, membership, and aggregate functions that can be easily customized with application-specific parameters. Applications specify which functions to use and the corresponding parameters in their subscription files as part of their quality requirements. A data-dissemination node, upon receipt of applications' quality requirements, invokes and instantiates filters accordingly.

Second, we allow applications to define domain-specific functions along any of the three dimensions of group-aware filters. Those functions can be specified with Uniform Resource Identifiers (URI) with which the group-aware filtering service at each node can dynamically upload from the application side. We use a programming model that supports inheritance, such as the Java language's class extension and polymorphism, to enable flexible extensions with well-defined filtering procedures and interfaces. For example, we allow each filter to extend the basic group-aware filter's *isAdmissible* method to apply domain-specific functions in candidate admission.

Third, our group-aware filtering service dynamically invokes pertinent group-aware filtering algorithms based on applications' specification. For instance, if the applications specify that the filter's candidate sets are dependent on one other, we invoke the per-candidate-set based algorithm, rather than the region-based algorithm.

Fourth, we expand the classic hitting-set problem and solutions for filters considering multi-degree candidacy within a candidate set. We define the multi-degree hitting set problems as follows.

Definition 6 *Given n sets of tuples, each set i needs to pick $pickDegree_i$ tuples as its outputs. The multi-degree hitting-set problem is to find $pickDegree_i$ tuples for each set i, such that the resulting output sets i $Output_i$ satisfy that $\sum |output_i|, 1 \leq i \leq n$ is minimum.*

We can prove the following property of the multi-degree hitting-set problem.

Axiom 3 *The multi-degree hitting-set problem is NP-hard.*

Proof: The hitting-set problem is a special instance of the multi-degree hitting-set problem when each set's $pickDegree$ equals 1. The hitting-set problem is NP-hard [25], and multiple-degree hitting-set problem is more general, or reducible to the hitting-set problem. Thus, multiple-degree hitting-set problem is also NP-hard. □

The greedy algorithm for the multi-degree hitting-set problem is slightly different from the one we defined before for the 1-degree hitting-set problem; when we find all candidate sets including (or being hit by) a selected tuple, for example, we do not remove or cover the candidate sets just yet. We need to update the number of tuples that have been chosen for output for each candidate set, and until the number of tuples has reached the required quantity of each set, we remove that candidate set from the next rounds of hitting process.

For filters with prescription for output selection other than "random," when deciding candidate sets, we need to apply an appropriate function to enforce the prescription. For example, if the prescriptive is "top" for k-degree candidacy, meaning that we need to pick the top-k tuples from each candidate set, we need to pre-determine the outputs by first sorting the candidates and eliminating tuples not belonging to the top-k. Notice that there might be more than k tuples if ties exist. During the greedy hitting process, those tuples not belonging to the top-k will not contribute to the tuples' group utilities. At each hitting process, beyond checking whether the degree of candidacy is satisfied, we make sure the selected tuples are still satisfying the top-k requirements (i.e., there should not be more than one tuple for each of the k ranks).

5.4 Evaluation

We devised a set of diverse group-aware filters and evaluated their basic performance with the NAMOS buoy data traces.

Table 5.1 lists the types of filters used for our testing. We denote a filter type with a type name followed by a set of parameters enclosed in a pair of parentheses. The first three types, $DC1$, $DC2$ and $DC3$, use the Delta-Compression (DC) theme and they vary from one another by how a candidate set is computed. $DC1$ filters monitor the change of a single attribute. It has three parameters: attribute name, delta and slack. $DC2$ filters monitor the change of the "trend" of an attribute. The trend reflects the change of an attribute over a unit of time. It has three parameters similar to those of $DC1$. $DC3$ filters monitor changes in the average of three attribute values. It has five parameters: three attributes used for averaging, and the delta and slack for delta compression. The last type SS (for Stratified Sampling) differs from the previous three types mainly in the second processing stage: the number of outputs (or sample rate) are determined by the candidate set's sample range. The sample range is the interval between the max and min value within a set of values. If the filter uses a fixed time interval to segment the time series, the sample range reflects the dynamics of the attribute within the interval. For applications sensitive to the dynamics of state change, it is reasonable to sample a time segment with high dynamics using a high sample rate. Thus, the parameters of SS filters are the attribute of interest, a threshold value that determines whether an interval is highly dynamic, a sample rate (percent of tuples) for more dynamic time segments, and a sample rate for less high dynamic intervals. Since it is hard for us to define a domain-specific function for selecting candidate sets, we here simply segment the time series with a time-based predicate. A sub-sequence of the data trace that has higher sample range means high dynamics of the "attrib" and it is reasonable for some applications to increase the sample rate during those time segments.

Filter type	Select candidates based on	Decide output
DC1(attrib, delta, slack)	change of attrib between delta--slack and delta+slack	choose any 1 tuple
DC2(attrib, delta, slack)	change of trend(attrib) between delta--slack and delta+slack	choose any 1 tuple
DC3(attrib1, attrib2, attrib3, delta, slack)	change of average(attrib1, attrib2, attrib3) between delta--slack and delta+slack	choose any 1 tuple
SS(attrib, timeInterval, threshhold, highSmplRt, lowSmplRt)	change of timeStamp within timeInterval	choose any n% of the tuples, where n=highSmplRt, if sampleRange(attrib) is no less than threshhold; or n=lowSmplRt, if sampleRange (attrib) is less than threshhold

Table 5.1: Types of group-aware filters for evaluation.

Group	Filter 1	Filter2	Filter 3
1	DC1(fluoro, 3.012, 1.506)	DC1(fluoro, 7.024, 3.012)	DC1(fluoro,5.000, 2.500)
2	DC1(thermo2, 0.0230,0.0115)	DC1(thermo2, 0.0460, 0.0230)	DC1(thermo2, 0.0315,0.0107)
3	DC1(thermo4, 0.0310, 0.0155)	DC1(thermo4, 0.0620, 0.0310)	DC1(thermo4, 0.0480, 0.0240)
4	DC1(thermo6, 0.0250, 0.0125)	DC1(thermo6, 0.0500, 0.0250)	DC1(thermo6, 0.0345, 0.0172)
5	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0600, 0.0300)	DC3(thermo2, thermo4, thermo6, 0.0452, 0.0226)
6	DC2(fluoro, 11.59, 5.79)	DC2(fluoro, 5.79, 2.89)	DC2(fluoro, 7.50, 3.75)
7	SS(thermo4, 1000, 0.1500, 50, 20)	SS(thermo4, 1000, 0.3000, 50, 20)	SS(thermo4, 1000, 0.2300, 50, 20)
8	DC1(thermo4, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	DC1(thermo5, 0.0300, 0.0150)
9	DC1(thermo4, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	DC2(fluoro, 3.00, 1.50)
10	DC1(thermo4, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	SS(thermo4, 1000, 0.1000, 90, 50)

Table 5.2: Specifications for ten groups of filters.

We made ten groups of filter specifications, as shown in Table 5.2. A specification is notated as the filter type followed by a set of parameters enclosed in a pair of parentheses. For delta-compression filters, the main parameters are the delta and slack value. For stratified sampling filters, main parameters consist of a threshold for determine whether a segment is important, a sample rate for important data sequences, a sample rate for less important data sequences, and time interval for segmenting the time series.

Each of the first seven groups contains three filters of the same kind; each of the remaining three groups contains three heterogeneous filters. Recall that each filter selects

data based on some feature of the tuple or on internal state computed on earlier tuples. The DC type filters select candidates based on the state change between two tuples. The Average State Change (ASC) of a time series is the averaged state change over the time series, that is, the average change in attribute value from tuple to tuple. We compute the ASC for the relevant attributes of each DC filter. We set the delta values no smaller than the ASC to prevent the filter from removing nearly no tuples, in which case filters may not be useful for compression. Specifically, we assigned ASC, 2·ASC and a value randomly generated between ASC and 2·ASC to be the delta values of the filters in the delta-compression groups. For example, we measured the ASC of thermo4 in the trace, which is 0.031 degree. Then we set the delta values of the filters in group 3 to be 0.031, 0.062 and 0.048 degree for each filter respectively. The slack value of each delta-compression filter was set to be 50% of the corresponding delta value. Setting the slack more than 50% might cause a tuple to be part of more than one candidate set for a filter.

Figure 5.2 shows the results. We compute the output ratio for each batch of 100 tuples and then compute the average and median of the output ratios across all batches. For eight out of the ten groups, the average output ratios were less than 80%, i.e. the group-aware filtering can reduce the bandwidth demand to 80% of the bandwidth demand of self-interested filters.

Table 5.3 shows the average CPU costs of the test, and Figure 5.3 shows the ratio of CPU time for group-aware filtering to CPU time for self-interested filtering. In general, the group-aware filtering is more complex than self-interested filtering, as it involves group coordination. It is not surprising, therefore, that some of costs were more than double. For groups with simple filters, such as $DC1$ and SS filters, the absolute costs were all below 35 ms per batch. For groups with more complex filters, such as $DC2$ and $DC3$, the cost increased both for group-aware filters and self-interested filters, due to the more complex computations. It may also be due to our unoptimized Java implementation. In all those

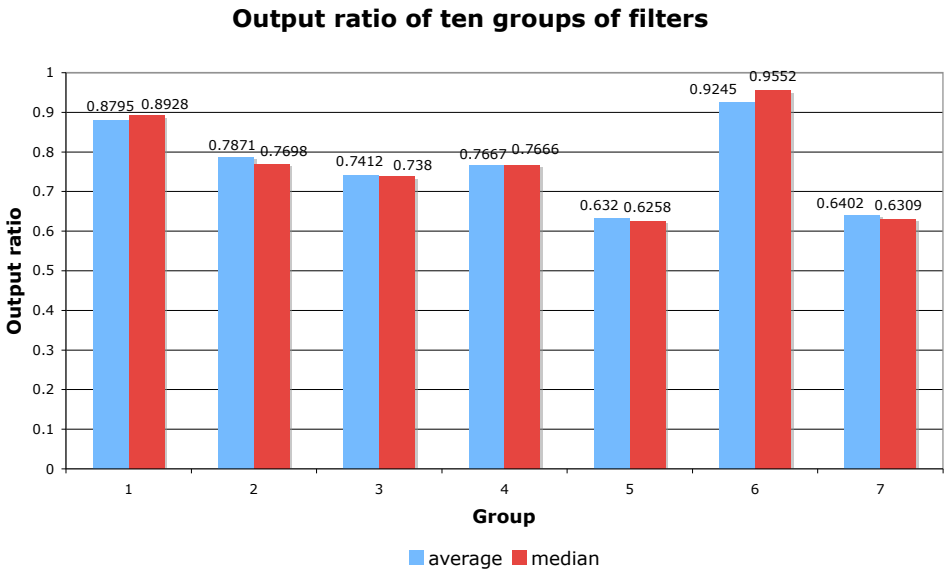


Figure 5.2: Benefit of group-aware filtering. Smaller output ratio implies better performance.

Group	Group-aware (ms)	Self-interested (ms)
1	28.03	10.46
2	28.86	10.4
3	22.20	13.36
4	26.45	14.95
5	386.00	260.44
6	684.75	334.50
7	31.00	14.90
8	21.92	11.82
9	32.29	14.35
10	41.76	16.30

Table 5.3: Average CPU cost per batch of 100 tuples.

cases, the CPU cost for processing a batch of 100 tuples was below 700 ms, that is, the cost for processing each tuple was below 7 ms, which is less than the data arrival rate (10 ms). Hence group-aware filtering will not cause congestion in these cases.

We also compared the incurred latency due to filtering with multicast latency. We set up 5 nodes on Emulab[1] in a Distributed Hash Table (DHT) ring for Solar's overlay multicast network. The links connecting the nodes were set to be 5 Mbps, a typical throughput for 802.11b networks. We measured the average latency of multicasting the batch of tuples between two nodes to be about 253.41 ms. Note that for wireless networks with effective bandwidth lower than 5 Mbps and with more nodes, the multicast latency should be much longer. For simple filters, the incurred filtering latency per batch (< 35 ms for 100 tuples) was negligible compared with the multicasting latency (≥ 253 ms). For complex filters, such as $D2$ and $D3$ type filters, whose incurred latency was longer than the multicasting latency, we can use timely cuts and output strategies to curb the latency.

[1] http://www.emulab.net is a cluster for distributed-systems research.

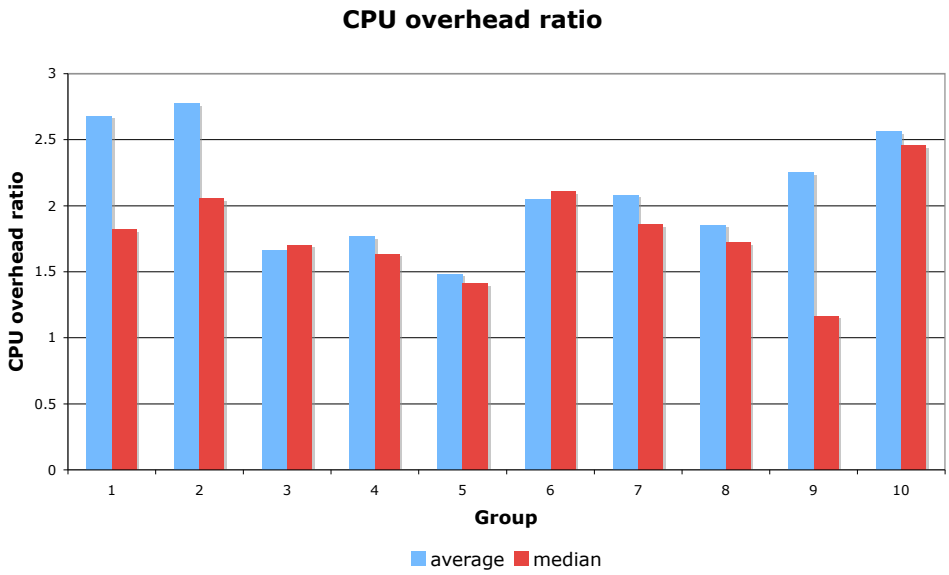


Figure 5.3: CPU Overhead ratios.

5.5 Discussion

The essence of group-aware filtering is to maximize data sharing among applications in an effort to minimize the size of the filter's output. The implication of reduced output has many potential benefits in resource-constrained systems for stream processing. Here we list a few scenarios where group-aware filtering can contribute to efficient resource management of the stream-processing systems.

5.5.1 Monitoring for emergency response

The first scenario is monitoring for emergency response; this is the scenario that originally motivated our group-aware filtering research.

For a general emergency-response scenario we imagine that an ad hoc mesh network results from wireless routers embedded in fire trucks, police cars, and ambulance, or are deliberately deployed to relay data (for example, wireless nodes hanging under a high-latitude balloon have been used for relaying weather-related data). We also imagine that sensor networks would be deployed to collect fine-grained data about the environment (fire, flood, gas clouds), locations of resources, or conditions of personnel. The raw sensor-data streams then register themselves as sources with the data-dissemination middleware system (such as Solar) that run on the wireless mesh nodes. Monitoring applications can then subscribe to the sources via the middleware for various command-and-control applications. For instance, in a fire scene, fire-prediction applications need sub-second data on chemical concentration, wind speed and wind direction. Safety monitoring applications for first responders may want to get per-second updates on the chemical concentration. A situation assessment application may require update of the chemical concentration at a different rate.

Our group-aware filtering approach integrated with Solar has been deployed as a middleware system for emergency-response research projects [48] in the past several years. For

example, in the spring of 2006, we used Solar in a larger-scale disaster-recovery scenario. We simulated a disaster where a train containing chlorine derailed in downtown Baton Rouge, Louisiana. In this scenario we disseminated a fast-rate data source that reports chlorine concentration in real-time to multiple command-and-control applications, including a fire-prediction application, a safety-assessment application for nearby responders, and a general situation-assessment application. Figure 5.4 shows one of those applications, a web portal that monitors the chlorine concentration and its spread in the nearby region of the disaster.

The three applications have different data-granularity requirements, so we deployed three group-aware delta-compression filters at the nodes where source data are disseminated. The source data was simulated according to a diffusion model that was carefully engineered for this scenario. The model considered many factors such as wind direction, wind speed, and the density of the sensors. The source produced a new reading every 10 ms. In this testing, we found that the group-aware filtering had a further bandwidth saving of about 15% over the self-interested data filters, and that overhead of the group-aware filtering processing was small: processing sixty tuples with the per-candidate-set based greedy algorithm took less than a quarter of a second. Reduced output of the group-aware filtering in this scenario helped to save precious network bandwidth in the data-dissemination system.

5.5.2 Multi-modal sensing with co-located sensors and imagers

Recent years have seen many advances in the technology of sensors and sensing platforms, and sensors span the spectrum of cost, form-factor, resolution, and functionality. Multi-modal sensing systems that mix low-fidelity, low-cost sensors with high-fidelity high-cost sensors in a sensing site provide an interesting balance between cost and functionality. Such

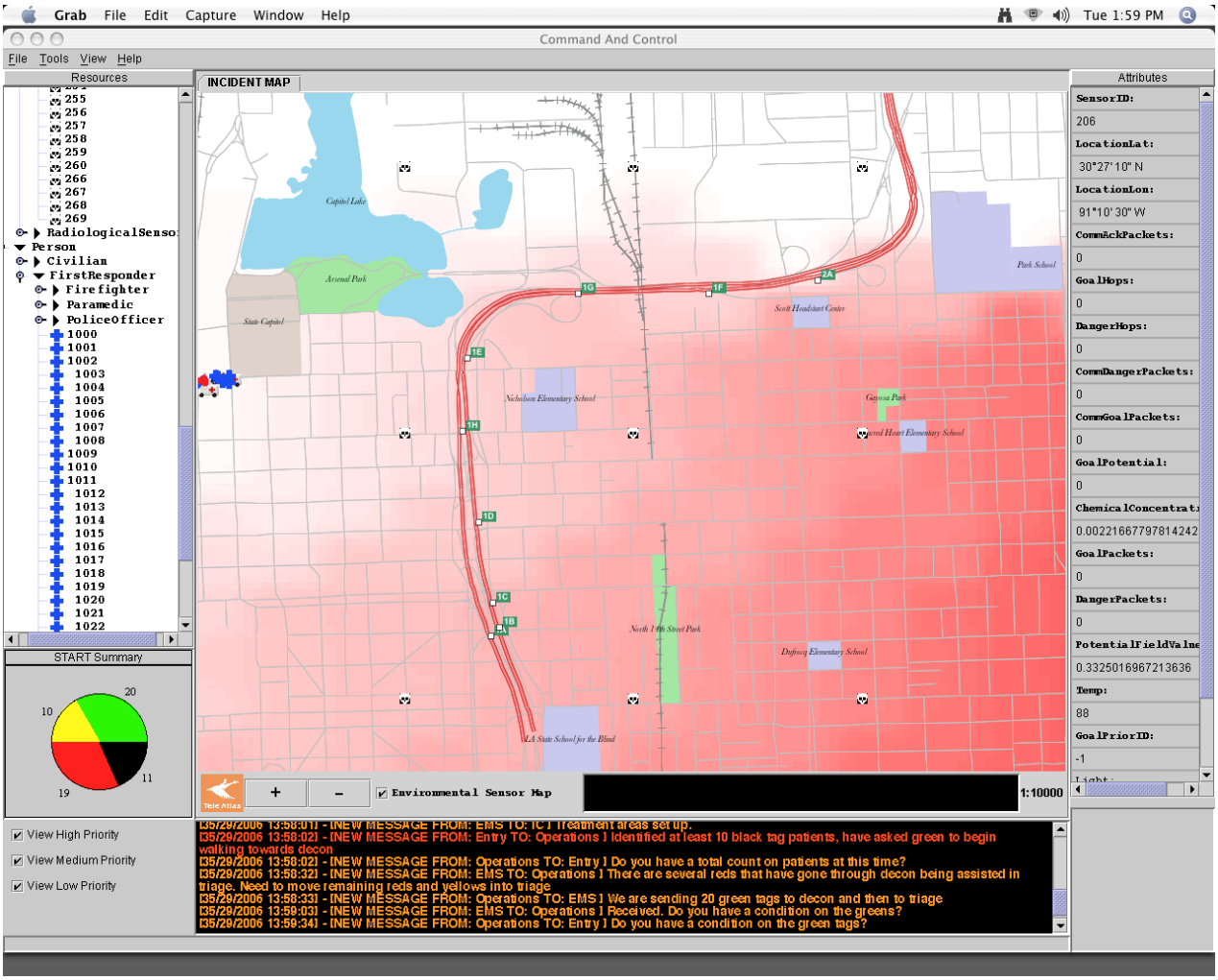


Figure 5.4: Chlorine monitoring in a train-derail disaster

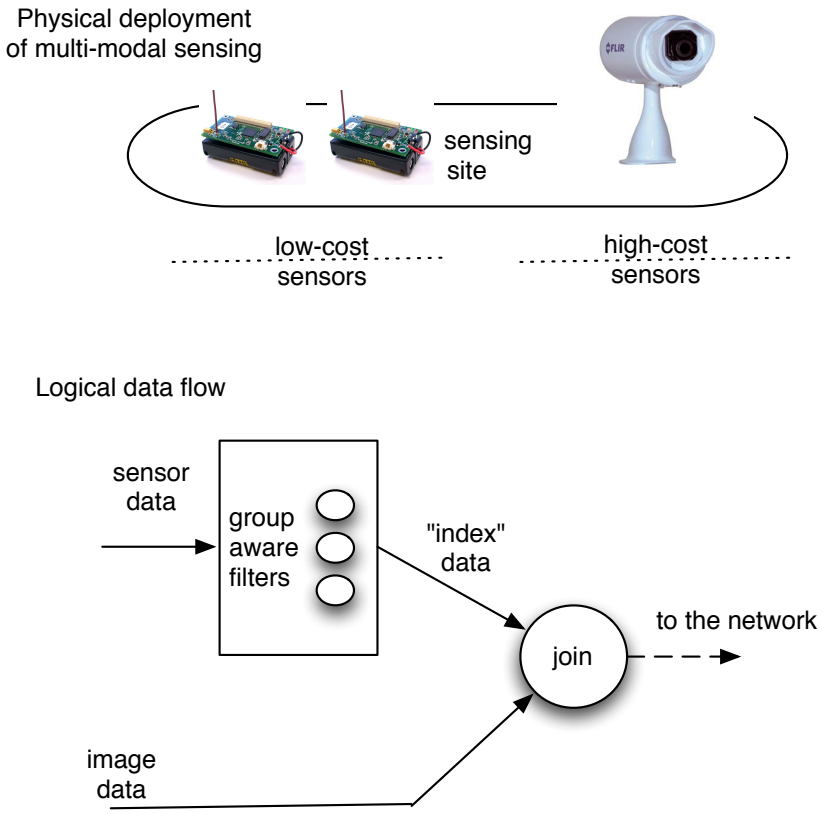


Figure 5.5: Multi-modal sensing with group-aware filtering.

multi-modal sensing strategies have been considered for vehicle-surveillance applications on a bridge [53], and a general multi-tier, multi-modal camera sensing framework [36, 37].

Given a multi-modal sensing environment, group-aware filtering can save precious system resources in several important ways. For example, Figure 5.5 shows a scenario of a wireless surveillance system in which low-cost sensors, such as motion or seismic sensors, are bundled with a high-cost imager and deployed at the same site for remote monitoring. We assume that the goal of reducing the bandwidth consumption for transmitting important camera pictures is important.

The low-cost sensors can sample the targetting environment at a high rate to serve multiple surveillance applications. Delta-compression filtering and stratified sampling are

both applicable for the surveillance application domain. Next, the output of the filters of the cheaper sensors are temporally correlated with the images taken by the high-resolution imagers to select the most informative images to send over the network to preserve network bandwidth. If we apply group-aware filtering to the filters, it will explore opportunities to reduce the output of the filters. The smaller the size of the output of the filters of the sensors, the smaller the number of the images selected to transport to remote applications. In this sense, we call the output from the filters of the low-cost sensors the "index" for selecting data from the imagers. In another scenario, imagine that we deploy such a sensor-imager bundle on a mobile robot for territory exploration; the indexing data may trigger cameras to take pictures. Thus it can potentially save the battery power needed for the robot to take the pictures and transmit them. It also saves space to store those images at the site in case of temporary network disconnection for delay-tolerant monitoring applications.

5.5.3 Sensor sampling for multiple queries

Group-aware filtering may also be applicable to data aggregation in sensor networks to support multiple queries. The idea is that each sensor provides a data source based on filtering conditions dictated by the queries it serves. Given multiple applications' differentiating filtering requirements, it will provide an opportunity to reduce data to be transmitted by leveraging group-aware filters at each sensor.

The overhead of running group-aware filtering may stress the CPU and memory at a sensor. With advances in sensor technologies, this can become less of an issue, and group-aware filtering provides another tool to help trade bandwidth and computation time in an effort to build a cheaper, longer-lasting sensor network.

5.6 Related work

Our group-aware filtering framework adopts many mechanisms from the state-of-the-practice of extensible systems. We discuss related work in software extensibility. We also briefly discuss the mechanisms to support applications to specify diverse quality requirements.

Extensibility is an important aspect of software systems to ensure compatibility with new implementation techniques and new capabilities to support new classes of applications. Many successful collaborative open source software developments [50, 10, 14, 27, 29] embrace a software architecture that promotes anarchic collaboration through extensions, while at the same time preserving centralized control over the interfaces. To achieve extensibility, many database systems consider availability of extensible data models, to allow for the introduction of new object types and operations, and extensible storage structure, to take advantage of special properties of stored data or operations to enhance performance. For example, EXODUS [56] uses a modular and modifiable system rather than a complete system to handle new application areas. Application-specific access methods, operation and version management layers can be constructed using the primitives provided by the storage system. Like our group-aware filtering framework, it also provides libraries of useful routines and rules for the extensible components of the system. STARBURST [45], from IBM, focuses on producing portable data-management systems to support a wide range of applications and to couple effectively to mainframes, servers, and workstations. It focuses on vertical distribution between host, server, and workstation. It also looks into the problem of how to integrate concurrency control and recovery, or user-managed data with DBMS-managed data. Our stream systems allow user-defined filters and provide libraries of customizable filters and functions for defining filtering needs.

Object-oriented programming provides several mechanisms for software extensibility: abstract data types, inheritance and polymorphism, and modular design. To take advantage

of those mechanisms, type theory for objects and their interactions help to aid checking and controlled derivation of programs and to support early binding of code bodies for efficiency [26]. Our prototype is implemented in Java, a platform-independent object-oriented language. It supports extensibility features with minimum complexity, compared with other OO languages, such as C++. It also provides compile-time type-checking facilities that we can leverage.

For communicating diverse filtering requirements, besides supplying self-defined functions, applications can supply predicates, logics or rules to declare properties of the data that will or will not satisfy the requirements. STREAM [2], from Stanford University, focuses on a declarative query language for continuous data streams by introducing extension clauses such as "sample rate" and "window". Aurora/Borealis [7] supports applications to define QoS functions for dynamic invocation of data-shedding operators. In sensor networks, multiple combinations of sensors and inference units deployed can answer the same query. Semantic Streams [70] let users to declare QoS constraints in CLP(R) notation to choose between logically equivalent inference graphs in sensor networks. An example of such QoS requirements is "I want to minimize the total number of radio messages." For group-aware filtering the most important information from applications' requirements is what data are treated as equivalent in quality. These declarative techniques are complementary to our current specification methods.

Chapter 6

Conclusion and Future Work

Data load management is the number-one challenge faced by stream-processing systems that support emerging monitoring applications that continuously collect, aggregate and disseminate high-volume high-rate information across a network. The traditional store-index-and-then-process data processing model no longer fits for processing fast and potentially infinite data streams, especially for data operations with a blocking effect on the data flow, such as join operators in relational algebra.

State-of-the-art stream-processing systems find approximate processing solutions that trade off data quality to save computational time and memory. They separate data processing from data transportation, based on the premise that the network resource is abundant compared with computation power. We recognize that, when multi-hop wireless mesh networks are used for data dissemination, the limitation on bandwidth is more severe than the limitations on computation power in managing the data quality perceived by the applications and the systems' scalability to the number of applications supported.

6.1 Contributions

The challenge of stream-processing systems that use slow wireless networks motivates us to develop approximate data processing methods that preserve network bandwidth. Our group-aware filtering approach is one of the first efforts to leverage approximate data processing for efficient data delivery. The main contributions of this dissertation are as follows.

- We make two key observations of properties of monitoring applications, based on which we propose a group-aware stream filtering approach that leverages approximate data processing to aggressively preserve network bandwidth for network-constrained data-stream systems.

- We thoroughly treat the group-aware stream filtering problem: we formally prove its NP-hardness, and provide an approximation-ratio-preserving segmentation scheme to apply heuristics-based algorithms that we develop for the problem.

- The framework we propose is quality-managed in that the algorithms ensure data timeliness and data granularity in addition to the goal of preserving network bandwidth.

- Our group-aware filtering framework is general and extensible for meeting a diverse range of data filtering needs. The idea of group-aware filtering is applicable to other services, such as smart sensing and efficient storage, beyond saving bandwidth.

- We evaluate our approach with a prototype system. The results, based on real-world data sets, show that the group-aware filtering can effectively save bandwidth with low CPU overhead compared with self-interested filtering. We also evaluate the effect of each algorithm on the time freshness of the data. We also discuss suitable parameters for using our approach and suggest possible extensions for future work.

6.2 Limitations and future work

Our group-aware filtering is a tool for exploring opportunities to maximize the data sharing among applications that belong to the same multicast group, where the goal is to save bandwidth. The overlap is intrinsically determined by data-quality requirements of the applications and the source data characteristics. Also, for continuously running filters, data characteristics may change over time, which affects the performance of group-aware filtering over time. In addition, when one of the filters in the group selects a large portion of data, the room for group-aware filtering to explore opportunities to save additional bandwidth is thin. Hence, one of the limitations of our work is that we need a more accurate online performance monitoring and prediction service for group-aware filtering, and to use more flexible strategies to decide when and how to use group-aware filtering, as in some situations the benefit of group-aware filtering may not be able to justify its computational overhead and its added latency to data. Future work to this end is to build a cost model of group-aware filtering, based on good prediction algorithms and on analysis of the factors that affect filtering performance. For situations where group-aware filtering does not affect bandwidth savings, we can dynamically disable group-awareness, and enable group-awareness in the filters when the predicted benefit is high. Also, our work has not addressed the utility of an output tuple to each individual filter. It may be the case that the compromise in data quality for group-aware filtering is not evenly distributed across all the filters. To this end, each application may want to specify more parameters to define the maximal tolerance of quality "skew" due to group-aware filtering. Then, the complexity of group-aware filtering will inevitably increase.

Our group-aware filtering assumes a medium-sized group to make the multicasting overhead reasonably small. When the size of the group becomes large, its high CPU cost may cause congestion at the input buffers of the filters. For this limitation we may in-

vestigate ways to group the filters into several smaller groups. But this surely leads to a reduced bandwidth-saving benefit of group-aware filtering. How to develop a cost model for guiding such a trade-off is a future work. Currently we group all the filters subscribing to a source, regardless of the network topology between the source and the applications. In scenarios where two applications share little of their routes from the source, there is little benefit to maximize the overlap of the two output streams. Hence, if we can group the applications' filters by how applications are clustered in the network, we may reduce the overall bandwidth needs even more.

Our framework supports a diverse range of filters, yet group-awareness may not be suitable for every filter, in that the filter may involve different or much more complex processing pattern than the two-stage group-aware processing. Also, some filters may require getting the whole candidate set before deciding the output, which prevents enforcing timely cuts. Developing method-rewriting techniques for turning filters into group-aware filters and devising property-testing methods will greatly help to tackle the problems.

For filters whose candidate sets have multi-degree candidacy, the semantics of enforcing timely cuts may be hard for applications to define. The complexity of defining semantics for timely cuts, and other operations in the execution of group-aware filtering algorithms, may be high. How to alleviate the application programmers' burden of writing dynamic specifications or policies is important to make group-aware filtering usable for real-life applications.

Our group-aware filtering is not a full-featured framework, as it does not address the security vulnerability it introduces in the system. Due to group-awareness, it creates vulnerability in "altruistic" filters to be affected by ill-mannered or malicious filters in the group. Group admission control is thus needed. Also due to the increased complexity of group-aware filtering compared to self-interested filtering, the system is more prone to fault and failure. How to ensure robustness and reliability is important for applying group-aware

filtering to real-life systems.

Our current evaluation is based on real-world data traces, yet the data quality requirements come from our speculation about future applications' needs. Hence, it is important future work to apply group-aware filtering in real applications and systems with real-life requirements. This approach will build a much stronger foundation for the evaluation. Also, there is a lot of room to explore novel applications with group-aware filtering. For instance, in a wireless surveillance system it is desirable to bundle a set of cheap sensors with high-resolution cameras and to deploy them at the same site for monitoring. We can imagine that different pattern-detection filters, for different safety analysis purposes, feed on the readings of the cheap sensors. The output of such a filter produces small indices to select the most relevant images to send over the network in order to save bandwidth. Applying group-aware filtering at the pattern-detection filters may help to produce even smaller indices to select data from bandwidth-demanding data sources to further save network bandwidth or the communication cost in battery-powered data sources.

The focus of our work is mainly at the data-processing aspect of group-aware filtering. Although group-aware filtering saves network bandwidth, it is not yet clear how it affects other aspects of network performance and how it could adapt to the dynamic conditions in a wireless network.

6.3 Conclusion

There is a well-recognized challenge faced by wireless data-dissemination systems: how to satisfy the high-volume data acquisition needs of emerging monitoring applications with bandwidth-limited wireless transport. Resolving this challenge requires an aggressive approach beyond traditional bandwidth-saving methodologies, such as multicasting and self-interested filtering. At the application level we exploit the the approximate nature of data

acquisition requirements and optimize the output for a group of data-sharing applications to reap further bandwidth savings with multicasting, while maintaining targeted data quality. We derive a general two-stage process for this "group-aware filtering" approach and our framework supports a wide variety of data-filtering needs.

Our evaluation, based on real-world data, shows encouraging results and reveals some key factors that affect the performance of group-aware filtering. Overall, group-aware stream filtering is likely to be a useful tool for tackling data-load challenges in wireless stream-processing systems.

Bibliography

[1] Hirotugu Akaike and Genshiro Kitagawa. *The Practice of Time Series Analysis*. Springer-Verlag, first edition, 1999.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: the Stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 665–665. ACM Press, 2003.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] Suresh Aryangat, Henrique Andrade, and Alan Sussman. Time and space optimization for processing groups of multi-dimensional scientific queries. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, pages 95–105, 2004.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[6] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634, 2002.

[7] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Mike Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4), January 2004.

[8] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–15, March 2004.

[9] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Sampling algorithms: lower bounds and applications. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing (STOC)*, pages 266–275, 2001.

[10] D. S. Batory and M. Mannino. Panel: Extensible database systems. *SIGMOD Record*, 15(2):187–190, 1986.

[11] Riccardo Bettati and Jane W.-S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 452–459, 1992.

[12] Barry Boehm, Prasanta Bose, Ellis Horowitz, and Ming June Lee. Software requirements negotiation and renegotiation aids. In *Proceedings of the 17th International Conference on Software Engineering (ICSE)*, pages 243–253, 1995.

[13] Andrew Campbell and Geoff Coulson. A QoS adaptive transport system: design, implementation and experience. In *Proceedings of the Fourth ACM International Conference on Multimedia (MULTIMEDIA)*, pages 117–127, 1996.

[14] Michael Carey and Laura Haas. Extensible database management systems. *SIGMOD Record*, 19(4):54–60, 1990.

[15] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 838–849, 2003.

[16] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, October 2002.

[17] M. Castro, M.B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE Computer Society Press, 2003.

[18] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, pages 111–122. Morgan Kaufmann Publishers Inc., 2000.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *Pro-*

ceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 668–668, 2003.

[20] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 263–274, 1999.

[21] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pages 246–255. ACM Press, 2004.

[22] Guanling Chen, Ming Li, and David Kotz. Data-centric middleware for context-aware pervasive computing. *Pervasive and Mobile Computing*, 4(2):216–253, April 2008.

[23] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.

[24] Reynold Cheng, Ben Kao, Sunil Prabhakar, Alan Kwan, and Yicheng Tu. Adaptive stream filters for entity-based queries with non-value tolerance. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 37–48, 2005.

[25] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[26] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programmimg. *ACM Computing Surveys*, 20(1):29–72, 1988.

[27] Mike Davis, Will O'Donovan, John Fritz, and Carlisle Childress. Linux and open source in the academic enterprise. In *Proceedings of the 28th Annual Conference on User Services (SIGUCCS)*, pages 65–69, 2000.

[28] R. M. C. de Keyser, P. Löhnberg, and H. B. Verbruggen. Applying adaptive control–problems and solutions: a survey. *Journal a*, 27(3):111–119, 1986.

[29] Roy T. Fielding. Software architecture in an open source world. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 43–43, 2005.

[30] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, Eugene Wu, Owen Cooper, Anil Edakkunni, and Wei Hong. Design considerations for high fan-in systems: The HiFi approach. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 290–304, 2005.

[31] Lukasz Golab and M. Tamer Ozsu. Issues in data stream management. In *ACM SIGMOD Record*, pages 5–14. ACM Press, 2003.

[32] Nen-Fu Huang and Whai-En Chen. RSVP extensions for real-time services in hierarchical mobile IPv6. *Mobile Networks and Applications*, 8(6):625–634, 2003.

[33] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 1–12. ACM Press, 2005.

[34] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods, and Algorithms*. IEEE, first edition, 2003.

[35] Benjamin Chi-Ming Kao. *Scheduling in distributed soft real-time systems with autonomous components*. PhD thesis, Princeton University, 1995.

[36] Purushottam Kulkarni, Deepak Ganesan, and Prashant Shenoy. The case for multi–tier camera sensor networks. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 141–146, 2005.

[37] Purushottam Kulkarni, Deepak Ganesan, Prashant Shenoy, and Qifeng Lu. SensEye: a multi-tier camera sensor network. In *Proceedings of the 13th Annual ACM International Conference on Multimedia (MULTIMEDIA)*, pages 229–238, 2005.

[38] Wang Lam. *Multicast data dissemination*. PhD thesis, Stanford University, 2005.

[39] Li Lao. *Service overlay networks for multicast applications*. PhD thesis, University of California at Los Angeles, 2006.

[40] Emmanuel Letier and Axel van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT)*, pages 53–62, 2004.

[41] Ming Li and David Kotz. Group-aware stream filtering. In *Proceedings of the Fourth Workshop on Wireless Ad hoc and Sensor networks*, pages 14– 22, June 2007.

[42] Ming Li and David Kotz. Group-aware stream filtering for bandwidth-efficient data dissemination. *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, Accepted for publication, 2008.

[43] J. W. S. Liu and K. J. Lin. On means to provide flexibility in scheduling. In *Proceedings of the second international workshop on Real-time Ada issues (IRTAW)*, pages 32–34, 1988.

[44] J.W.S. Liu, S. Natarajan, and K.J. Lin. Imprecise results: utilizing partial computations in real-time systems. pages 210–217. IEEE, 1987.

[45] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to Starburst: objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, 1991.

[46] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 49–60. ACM Press, 2002.

[47] David Maier, Peter A. Tucker, and Minos Garofalakis. *Stream Data Management*. Springer, first edition, 2005.

[48] S. McGrath, E. Grigg, S. Wendelken, G. Blike, M. De Rosa, A. Fiske, and R. Gray. ARTEMIS: A vision for remote triage and emergency management information integration. *http://www.ists.dartmouth.edu/projects/frsensors/artemis/*.

[49] Don P. Mitchell. Consequences of stratified sampling in graphics. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 277–280. ACM Press, 1996.

[50] Audris Mockus, Roy Fielding, and James Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering Methodology*, 11(3):309–346, 2002.

[51] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, pages 35–46, 2000.

[52] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 563–574, 2003.

[53] Jeffrey Ploetner and Mohan M. Trivedi. A multimodal approach for dynamic event capture of vehicles and pedestrians. In *Proceedings of the 4th ACM International Workshop on Video Surveillance and Sensor Networks (VSSN)*, pages 203–210, 2006.

[54] Venkatesh Raghavan, Elke A. Rundensteiner, John Woycheese, and Abhishek Mukherji. FireStream: Sensor stream processing for monitoring fire spread. In *Proceeding of the 23rd International Conference on Data Engineering (ICDE)*, pages 1507–1508. IEEE, 2007.

[55] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Third International Workshop on Networked Group Communication (NGC 2001)*, November 2001.

[56] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. *SIGMOD Record*, 16(3):208–219, 1987.

[57] William N. Robinson and Vecheslav Volkov. Supporting the negotiation life cycle. *Communications of the ACM*, 41(5):95–102, 1998.

[58] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 2001 International Middleware Conference*, pages 329–350. Springer-Verlag, November 2001.

[59] Daniela Rus. Networked robotics. *http://robotics.csail.mit.edu/projects/rus.pdf*.

[60] Steven Skiena. *The Algorithm Design Manual*. Springer-Verlag, second edition, 1998.

[61] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[62] John Strassner. *Policy-Based Network Management: Solutions for the Next Generation*. Morgan Kaufmann, first edition, 2003.

[63] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE)*, 1998.

[64] Guangming Tan, Ninghui Sun, and Guang R. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 135–144, 2007.

[65] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases (VLDB)*, pages 309–320, 2003.

[66] Steven K. Thompson. *Sampling*. John Wiley Sons, second edition, 2002.

[67] Upkar Varshney. Multicast over wireless networks. *Communications of the ACM*, 45(12):31–37, 2002.

[68] Jack Walicki and John D. Laughlin. Operation scheduling in reconfigurable, multi-function pipelines. In *Proceedings of the 20th Annual Workshop on Microprogramming (MICRO)*, pages 80–87, 1987.

[69] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[70] Kamin Whitehouse, Feng Zhao, and Jie Liu. Automatic programming with semantic streams. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 290–291, 2005.

[71] Nicholas J. Yeadon, Francisco Garcia, David Hutchison, and Doug Shepherd. Filters: QoS support mechanisms for multipeer communications. *IEEE Journal of Selected Areas in Communications (IJSAC)*, 14(7):1245–1262, 1996.

[72] Pamela Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.

[73] Yuanyuan Zhao and Rob Strom. Exploiting event stream interpretation in publish-subscribe systems. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, 2001.

[74] Hu Zhou and Suresh Singh. Content based multicast in ad hoc networks. In *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc)*, pages 51–60, 2000.

[75] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.